PROCEEDINGS OF THE

FIRST WORKSHOP ON

LOGICAL FRAMEWORKS

Antibes

May 1990

Eds G. Huet & G. Plotkin

Contents

Foreword

This document is the proceedings of the First Annual Workshop on Logical Frameworks, funded by the Esprit Basic Research Action 3245 "Logical Frameworks: Design, Implementation and Experiment".

This workshop was organised in Antibes by Gilles Kahn, with the help of INRIA's Service des Relations Extérieures, from the 7th to the 11th of May, 1990.

These preliminary proceedings have been collected from LaTeX sources, using electronic mail. Some attempt has been made to edit a uniform document, but the variety of styles of the various authors precluded a completely satisfactory result. We regret that the insistance on LaTeX sources prevented some papers, such as the ones from the Paris7 group, to be included in these proceedings.

⋆Logical Frameworks⋆

Esprit Basic Research Action
First Annual Workshop
Antibes, May 7 - 11, 1990

Final Program

Monday, May 7th, at INRIA Sophia-Antipolis

Demonstration Session 1. 9:30 am Chair Rod Burstall

Presentations of the demos (*conference room, 10 mn. max each*)

10:30 Parallel Computer Demonstrations

- **Lawrence C. Paulson and Tobias Nipkow.** Isabelle.

  Abstract. Isabelle is an interactive theorem-prover that supports a variety of logics. It represents rules as propositions (not as functions) and constructs proofs by combining rules. These operations constitute a meta-logic (or 'logical framework'), in which the object-logics are formalized. Isabelle is now based on higher-order logic — a precise and well-understood foundation. Examples illustrate the use of this meta-logic to formalize logics and proofs. Axioms for first-order logic can be shown sound and complete. Backwards proof is formalized by meta-reasoning about object-level entailment. Higher-order logic has several practical advantages over other meta-logics. Many proof techniques are known, such as Huet's higher-order unification algorithm.

- **Frank Pfenning.** Logical Frameworks and Logic Programming.

- **Lennart Augustsson, Thierry Coquand & Bengt Nordström.** Implementation of Another Logical Framework.

- **Laurent Théry.** Using Centaur as a front-end for any proof system.

- **Leen Helmink.** Proof Construction in Generalised Type Systems.

12:30 Discussion of the demos

Lunch 1 pm

Demonstration Session 2. 2 pm Chair Lennart Augustsson

Presentations of the demos (*conference room, 10 mn. max each*)

2:40 pm Parallel Computer Demonstrations

- **Matt Kaufmann.** The Boyer-Moore Theorem Prover and some of its extensions.

  Abstract. The Boyer-Moore theorem prover is a program written by Bob Boyer and J Moore to check theorems in a quantifier-free, first-order logic that resembles Pure Lisp. The theorem prover has been used to check substantial theorems in a number of areas. It is perhaps known best for its ability to discover induction schemes automatically, a limited implementation of metatheoretic extensibility, and an integrated linear arithmetic procedure. However, much of its utility derives from more mundane considerations such as extremely careful engineering, a simple logic, and a basic user interface that allows the user to "program" the rewriter.

  An extension of the Boyer-Moore prover is PC-NQTHM ("PC" for "proof-checker", "NQTHM" being the name of the Boyer-Moore theorem prover). That system, authored by Matt Kaufmann, extends the functionality of the Boyer-Moore prover by allowing lower-level interaction by the user as well as direction in the style of the "tactics" of systems such as LCF and Nuprl.

  In this demo I will work through a simple, but not quite trivial, example using the Boyer-Moore theorem prover and PC-NQTHM. People will see both some strengths and some weaknesses of these systems. My hope is that this introduction, although brief, will nevertheless give people a taste of how people really use these systems on more serious examples. If time permits I will also demonstrate some more recent extensions of these systems.

- **Yves Lafont.** Interaction nets.

  Abstract. Interaction Nets generalise Girard's Proof Nets of Linear Logic, providing a simple framework for distributed computing, where programs are like logics, processes are like proofs, and execution consists in cut elimination. We shall visualise some typical computations.

  Our system is implemented on top of CAML interfaced with XWINDOW (using the Virtual Bitmap of LE_LISP) for graphics and mouse interaction.

  This demonstrates that Interaction Nets are well adapted to step by step visualisation of computations. This should be especially interesting for teaching purposes.

- **Randy Pollack.** LEGO: An Interactive Proof Checker

  Abstract. Lego is a proof checker for several type theories related to the Calculus of Constructions, intended for research in machine-checked mathematics. Examples checked in Lego include the binomial theorem for rings, elementary topology, the deduction theorem for first order logic, derivation of sorting programs, and Coquand's proof of Girard's paradox.

  This work owes much to many people, most immediately Th. Coquand, R. Harper, G. Huet, and Z. Luo.

  The demonstration will show the following features of Lego:

  Type theories Lego supports the Edinburgh Logical Framework (LF), the Pure Calculus of Constructions, and Luo's Extended Calculus of Constructions (ECC) with a predicative hierarchy of cumulative universes and Sigma Types.

Refinement and Direct Proof The principal means of constructing proofs is interactive proof refinement by resolution. There are also features supporting direct bottom-up proof, and mixing of refinement and direct proof.

Definitions Each of the type theories is extended with a notion of 'definition'. These definitions serve both for notational extension and (using the propositions as types paradigm) for formalizing a context of proved theorems.

Inferred Polymorphic Instantiation All of the supported type theories have explicit polymorphism. Lego has an algorithm to infer some explicit polymorphic instantiation.

Typical Ambiguity Lego has an algorithm to infer universe numbers, thus providing 'typical ambiguity'. Lego also implements the 'let polymorphism' this typical ambiguity induces. To the user this system looks very like '`Type:Type`', although the typechecker maintains universe stratification.

Among the aspects of mathematical reasoning we are not satisfied with in Lego are equality reasoning and manipulation of 'theories'. For the future, I am working on a new implementation with a simpler and more uniform notion of refinement, and supporting Generalized Type Systems extended with Sigma Types and cumulativity.

Lego is available from LFCS, Computer Science Department, The King's Buildings, University of Edinburgh EH9 3JZ, UK.

- **Alan Smaill.** The Oyster-Clam system.

  Abstract. The Oyster system is a reimplementation of the NuPRL verion of Martin-Lof type theory, using Prolog both as the implementation language and the tactic language. The Clam system operates by reasoning at the meta-level about specification of Oyster tactics so as to construct an appropriate sequence of tactics for a given goal.

4 pm Discussion of the demos and tea

Demonstration Session 3. 4:30pm

Presentations of the demos (*conference room, 10 mn. max each*)

5 pm Parallel Computer Demonstrations

- **Christine Paulin & Benjamin Werner.** Extracting and Executing Programs Developed in the Inductive Constructions System.

  Abstract. Our ultimate goal is the realization of an environment for the development of certified correct programs.

  The Inductive Constructions System in an implementation of the Calculus of Constructions extended with inductive types. It has been developed, in the CAML language, by the Formel team at INRIA. The Constructions system is an interactive proof development environment, in which it is possible to define propositions and types, to check theorems or build proofs backwards, using tactics.

  We use the Calculus of Constructions for developing correct programs, seen as the computational meaning of constructive proofs of existential specifications. We eliminate the part of

the proof without computational content; then the resulting term is translated into a program written in an intermediate language FML. At this step, we use the LML compiler (a lazy ML version, developed by the Göteborg site) for the program execution itself. FML is a first step towards a reasonably efficient programming language, adapted to the execution of extracted programs.

The demonstration should illustrate the characteristics of the system, through the main development steps of the well-known quicksort algorithm.

- **David Carlisle.** A Type Theory implemented in the Logical Theory of Constructions.
- **Herbert Sander.** An implementation of the μ-calculus in Isabelle.

6pm Discussion of the demos and general discussion, moderated by G. Kahn

Can we share some development effort, how?

7pm Bus to go back to the hotel

Tuesday, May 8th

Session 1. 9 am Chair Gordon Plotkin

- **N. de Bruijn.** A plea for weaker frameworks.

  Abstract. It is to be expected that logical frameworks will become more and more important in the near future, since they can set the stage for an integrated treatment of verification systems for large areas of the mathematical sciences (which may contain logic, mathematics, and mathematical constructions in general, such as computer software and computer hardware). It seems that the moment has come to try to get to some kind of a unification of the various systems that have been proposed.

  A verification system consists of
  (i) a framework which defines how mathematical material (in the wide sense) can be written in the form of "books", such that the correctness of these books is decidable by means of an algorithm ("checker"),
  (ii) a set of basic rules ("axioms") that the user of the framework can proclaim in his books as a general basis for further work.

  For several reasons (theoretical, practical and sociological) it may be recommended to keep the frameworks as simple as possible and as weak as possible, and to prefer frameworks belonging to a hierarchy in which the various specima can be easily compared. There is, however, the opposite trend to put more power in the framework, since it is there that one gets the idea that things can be done automatically. In particular this may mean that much of what is achieved by "book equality" (the kind of equality that is to be considered as a mathematical proposition and for which proofs have to be provided by the user) can be shifted to "definitional equality" (based on reductions in the framework that can be handled automatically). But needless to say this tendency to provide the framework with facilities that depend on local and personal preferences may lead to an undesirable variety of diverging frameworks.

When one uses arguments of economy in order to put more power into the framework one forgets that it is neither very hard to put some automation in the task of writing books. We can invent abbreviational facilities that may result in automatic writing of parts of the books (parts that the user even does not need to see if he does not want to). In particular this may refer to handling trivial equalities like those we need when dealing with pairs and cartesian products. This need for automatic textwriting will turn up in many other situations too. It happens every now and then, when we want to economize on writing things that we consider as part of our subconcious thinking. It does not seem to be the right thing to enrich the framework at every occasion of that kind.

The general lambda-typed lambda calculus (in particular in the form of Delta-Lambda) seems to be a good candidate for the top of a framework hierarchy, and some of its more restrictive sub-systems might get a place as well. In particular there are the Automath-like systems of which the main characteristics are the following: By means of contexts we can deal with variables typed by anything we like (and that means a general kind of quantification), but abstraction is allowed only for variables of degree 3 (typed by some A that is typed by 'type' or some other expression of degree 1).

The early Automath versions AUT68 and AUT-QE were very advanced for their time (around 1968) in the sense that the frameworks were light and that almost all logical and mathematical knowledge was to be developed in the books. But both AUT68 and AUT-QE had a feature called type-inclusion that prevented them from being sub-systems of the general lambda calculus. In AUT68 this could be interpreted as making no distincions at all between expressions of degree 1: a typing "A:type" meant that A was an expression of degree 2. So what should have been written as "F:[x:B]type" according to the rules of the lambda calculus was just recorded as "F:type". This step has been called type inclusion: "[x:B]type" is included in "type". It means sacrifice of information: if we know "F:[x:B]type" then we know that the type of F is an expression of degree 1 that starts with the abstractor [x:A], and if we apply type inclusion that information is given up. AUT-QE is different in the sense that we are allowed to keep that information if we wish. If we know "G:[x:B][y:C]type" then we are free to reduce it to "G:[x:B]type" or to "G:type", but we may also leave it as it stands. AUT-QE had AUT68 as a sub-language, but Zucker's AUT-PI was different. It avoided type inclusion, but because of other features it was still not embeddable in lambda calculus. On the other hand AUT-QE-NTI of 1978 (NTI stands for "no type inclusion") is a pure lambda calculus system, embeddable in DeltaLambda. It seems to be relatively weak since it allows abstraction and quantification only over types A with A:type, but by means of axioms we can get about to the same expressive power as AUT-QE and AUT-PI. One possible set of axioms for AUT-QE-NTI refers to the universal quantifier All, or to its twin brother Pi that forms cartesian products. In a context A:type, Q:[x:A]type we postulate as a primitive All(A,Q):type; if in that context we have moreover y:All(A,Q), we have a further primitive Ax1(A,Q,x):<x>Q, and if in that same context we have z:<x>Q, we postulate Ax2(A,Q,x):All(A,Q). These Ax1 and Ax2 permit us to jump up and down from Q to Ax(A,Q), and that mimics the type inclusion feature without frustrating the typig rules of the lambda calculus. Another set of axioms in AUT-QE-NTI organizes the Sigma operation of AUT-PI. In the context A:type, Q:[x:A]type it postulates the type Sigma(Q), that can be imagined as being the type of all pairs p,q with p:A, q:<p>Q. With the terminology of telescopes it can be described as a type Sigma(A,Q) that enables us to replace the telescope [x:A][y:<x>Q] of length 2 by the telescope [w:Sigma(A,Q)] of length 1. Actually this set of primitives was used extensively in AUT68 in order to treat sets (subtypes) as types. We can also mention that eta reduction, a feature of several languages, can easily be replaced by a set of book axioms. A further candidate for being shifted from the framework to book axioms is the matter of admitting more than 3 degrees. It is to be expected that this feature can be mimicked efficiently by means of axioms over a framework that handles only 3 degrees.

The author thinks that both DeltaLambda and its much more restrictive sub-system AUT-QE-NTI can play a role in the unificaton of verification systems, in the sense that they can help to implement other systems by means of sets of axioms.

- **Peter Aczel & Nax Mendler.** A Type Theory implemented in the Logical Theory of Constructions.

  Abstract. The first part of the talk will briefly review the LTC (Logical Theory of Constructions) framework, as presently conceived. The framework is being implemented in Isabelle. It is a framework of interpreted languages (i.e. theories each with a standard interpretation) built on top of an extension of intuitionistic first order logic with equality.

  The language LTC_0 has an explicit lazy evaluation relation for a small functional programming notation and a relation for full natural number evaluation, both inductively defined in the standard interpretation. The language LTC_1 uses two additional unary predicates to express internal notions of proposition and truth that reflect the language LTC_0 in roughly the same way that first order logic is internally reflected in a Frege structure. LTC_ω iterates the reflection procedure ω times and has an infinite hierarchy of levels of proposition.

  In the second part of the talk three Martin-Löf-style type theories will be presented, TT_0, TT_1 and TT_ω, where TT_0 is a type theory without type universes, TT_1 has one type universe and TT_ω has a cumulative hierarchy of type universes. Then we give interpretations of each in the corresponding LTC theory. The novel style of the natural deduction rules for the type theories was chosen to simplify this interpretation.

- **Stefano Berardi.** Girard's normalization theorem in LEGO.

  Abstract. I will report in this talk:
  i) my attempt to formalize in LEGO a proof of Girard'S Normalization Theorem inside the Pure Calculus of Constructions;
  ii) some pragmatic considerations about possible developments of computer-aided proofs, suggested by this experience.

Coffee Break

Session 2. 11am

- **Thierry Coquand and Peter Dybjer.** Syntax and Semantics of Inductive sets.

- **Lennart Augustsson, Thierry Coquand, Lena Magnusson, and Bengt Nordström.** Another Logical Framework.

Lunch 12:30 pm

Session 3. 2 pm Chair Per Martin Löf

- **Peter Schroeder-Heister.** The role of elimination inferences in a structural framework.

  Abstract. It is claimed that I- and E-inferences operate at different levels. Whereas I-inferences express logical rules in the genuine sense, E-inferences are more like structural inferences. This is blurred by certain features of intuitionistic logic, but becomes obvious when subsystems of intuitionistic logic are considered.

- **Lars Hallnäs.** Models of partial inductive definitions.

  Abstract. Focusing on the structure of (partial) inductive definitions one may view such definitions as type systems for simply typed lambda calculus with abstract data types. This can be seen through a standard Curry-Howard interpretation of these definitions. Such a lambda calculus (or system of natural deduction) gives a rather natural interpretation of a definition and with suitable "modifications" one can see them as models with a rich structure that mirrors the intensional structure of a definition. Such models can be used as a basis for the classification of objects in a domain with respect to a given definition, for instance the "logical" status of an object in the context of the given definition. The modification we have in mind concerns the interpretation of abstraction. We will consider a kind of completion of a "flat" interpretation, i.e. abstraction and substitution will be given as local operations.

- **Paul Rozière.** Admissible rules in intuitionistic logic.

  Abstract. A rule is said "admissible" in a given logic if it doesn't generate new theorems when we add it to the rules of this logic. In classical propositional logic, every admissible rule is derivable. In intuitionistic propositional logic (IPC) there known counterexamples. We prove that in large fragments of IPC, every admissible rule is derivable. Then we propose a complete axiomatization and a decidable characterization of admissibility for IPC.

Tea Break

Session 4. 4:30 pm Chair Gérard Huet

- **Alan Bundy.** The Use of Proof Plans for Program Synthesis.

  Abstract. An overview of our work on guiding inductive proofs as applied to program synthesis using Martin Löf's Type Theory.

- **Rod Burstall.** An approach to program development in calculus of constructions.

7:15 pm. Methodological Panel, Jean-Yves Girard, discussant. Logic or Logics?

Wednesday, May 9th

Session 5. 9 am Chair Gilles Kahn

- **Robert L. Constable.** Transforming a classical proof of Higman's lemma into a constructive one.

- **Hugo Herbelin.** A-translation and its application to the synthesis of algorithms.

  Abstract. A-translation was used by Friedman to show that in formal number theory, every classically provable formula $\forall x \exists y \phi(x, y)$ with ϕ decidable is provable intuitionnistically. This can be extended to some others systems, even of second order.

  This is relevant when it comes to finding a constructive contents of some only classically proved theorems from which the constructive part isn't obvious. For example, Gabriel Stolzenberg tried with this method to extract a constructive proof from the classical proof of the Higman's lemma. The importance of this proof is justified because first of its logical complexity and second of the use of impredicativity.

  This is relevant too for extracting algorithms. We can develop "A-A-proofs", that are quite close to proof in classical logic, but are constructive, and such that one can extract an algorithm from them.

  What we do is related to these two considerations. We will first describe rapidly the A-translation we will use, following G. Stolzenberg's work. Then we will look at what the A-translation gives on some typical classical proofs. We then study more precisely what the A-translation of a restriction of the Higman's lemma gives as constructive proof and as extracted program.

- **Gabriel Stolzenberg.** Seeing Constructions.

Coffee Break

Session 6. 11am

- **Matt Kaufman.** System verification, Texas style.

  Abstract. Even if we prove software correct, what assurance do we have that the machines that execute the software will respect its semantics? Researchers at the University of Texas and Computational Logic, Inc. have addressed this problem by using the Boyer-Moore logic and theorem prover to formalize and prove the correctness of computing systems. This talk will summarize accomplishments of this group in proving correctness theorems for a compiler from micro-Gypsy (a high-level language) to Piton (a somewhat lower-level language), a link-assembler from Piton to FM8502 (a machine-language interpreter), and a correspondence from FM8502 to to a register-transfer model of hardware. These theorems combine to demonstrate the correctness of an application program in a high-level language with respect to a binary image on a microcoded machine.

- **Martin Coen.** Interactive Program Synthesis in ZF set theory.

  Abstract. Program synthesis may be regarded as a theorem proving task: the program to be synthesized is a "logical variable" which is instantiated during the proof of its specification. This talk will describe an attempt to produce a logic for program synthesis by defining a simple programming language of total functions within ZF set theory. The resulting framework allows reasoning about programs using well-founded recursion and induction, and has been used to derive a unification algorithm. The work has been carried out on machine using Larry Paulson's theorem prover Isabelle.

Lunch 12:30 pm

Session 7. 2 pm Chair Jean-Yves Girard

- **Jean Gallier.** Strong Normalization proofs: From system F to the calculus of constructions.

  Abstract. Girard's original method and its refinement due to Coquand for proving strong normalization in various typed lambda calculi will be discussed. The culmination of the method will be a proof of strong normalization for the calculus of constructions that does not use infinite contexts. Various technical difficulties present in earlier proofs will be discussed.

- **Michel Parigot.** Proofs of strong normalisation.

  Abstract. We discuss a way of proving strong normalisation (SN) by syntactic interpretations:
  (1) Interpretation of a non typed system into a typed system: we deduce in this way SN for some laballed lambda-calculus from SN for the simple typed lambda-calculus.
  (2) Interpretation of a typed system into itself: we deduce in this way SN from weak normalisation, for various typed system (including F).

- **Anne Salvesen.** The Church-Rosser theorem for LF with beta/eta reduction.

Tea Break

Session 8. 4:30 pm Chair Mario Coppo

- **Leen Helmink.** Proof Construction in Generalised Type Systems.

  Abstract. A method has been developed for proof construction in Generalised Type Systems (GTS), thus offering a well-founded and powerful underlying formalism for a proof development system. It combines the advantages of Horn clause resolution and higher order natural deduction style theorem proving. The method was originally developed for the Calculus of Constructions, but has meanwhile been generalised to GTS's with the help of Henk Barendregt. An interactive proof environment that implements the method has been developed in the framework of Esprit project 1222 Genesis. This system, named "Constructor", demonstrates the power and efficiency of the method. It has constructed many higher order proofs in different versions of Type Theory.

- **Gilles Dowek.** First design of a Mathematical Vernacular.

  Abstract. We present an attempt at a mathematical language (called Mathematical Vernacular) which is at the same time usable and precisely described. This language uses the Calculus of Constructions as a theoretical base. The main idea is to get rid of proof-terms and write demonstrations of theorems as sequences of propositions, such that a proof of each proposition is obvious when previous propositions are known. A proof-synthesis system is developed in order to check that a sequence of propositions is indeed a demonstration.

9:30pm. Business meeting: Editing of the Proceedings. Chaired by G. Huet.

Thursday, May 10th

Session 9. 9 am Chair Peter Aczel

- **Susumu Hayashi.** ATT: A Type Theory for optimized Curry-Howard Isomorphism.

Abstract. ATT (**A** **T**ype **T**heory) is introduced. ATT is a variant of Martin-Löf's type theory (**ITT**). It is defined as

$$\begin{array}{rc} \mathrm{ATT} = \mathrm{ITT} - & \{\Pi x \in A.B,\ \Sigma x \in A.B\} \\ + & \{\cup x \in A.B,\ \cap x \in A.B\ \ A \times B\} \\ + & \{\text{singleton type}\} \end{array}$$

ATT is used to optimize extracted programs by Curry-Howard isomorphism. More exactly, it gives a way to write specifications (as types) of programs without unnecessary codes, which is inevitable in the usual Curry-Howard isomorphism. I will use ATT to explain some optimization techniques for program extraction, e.g., subset type is definable in ATT. Impredicative version of ATT, which is similar to CC and AF_2, will be considered.

- **Nax Mendler.** Subset Types and Quotient Types via Coequalisers.

  Abstract. It is often advantageous to think of a Martin-Löf-style type theory as the internal language of a locally cartesian closed category. Then semantics can serve to guide us to proof rules. This note is an exercise in working from a categorical property (the existence of coequalizers) to arrive at the type theory rules for quotient types. We use the quotient types to define a "squash" type and then a subset type.

- **David Pym.** Investigations into Proof-search in the $\lambda\Pi$-calculus.

  Abstract. We present a series of proof systems for LP: a theory of *first-order dependent function types*. The systems are complete for the judgement of interest but differ substantially as bases for algorithmic proof-search. Each calculus in the series induces a search space that is properly contained within that of its predecessor. The LP is a candidate *general logic* in that it provides a metalanguage suitable for the encoding of logical systems and mathematics. Proof procedures formulated for the metalanguage extend to suitably encoded object logics, thus removing the need to develop procedures for each logic independently. This work is also an exploration of a *systematic* approach to the design of proof procedures. It is our contention that the task of designing a computationally efficient proof procedure for a given logic can be approached by formulating a series of calculi that possess specific proof-theoretic properties. These properties indicate that standard computational techniques such as *unification* are applicable, sometimes in novel ways. The study below is an application of this design method to an intuitionistic type theory. Our methods exploit certain forms of *subformula property* and *reduction ordering* — a notion introduced by Bibel for classical logic, and extended by Wallen to various non-classical logics — to obtain a search calculus for which we are able to define notions of *compatibility* and *intrinsic well-typing* between a derivation ψ and a substitution σ calculated by unification which closes ψ (a derivation is closed when all of its leaves are axioms). Compatibility is an acyclicity test, a generalization of the *occurs-check* which, subject to intrinsic well-typing, determines whether the derivation ψ and substitution σ together constitute a *proof.* Our work yields the (operational) foundations for a study of *logic programming* in this general setting. This potential is not explored here.

Coffee Break

Session 10. 11am

- **Frank Pfenning.** Logical Frameworks and Logic Programming (joint work with Ken Cline).

  Abstract. We discuss design and implementation of Elf, a "logic" programming language based on LF. Elf gives an operational interpretation to LF types in a manner analogous to the way Prolog gives an operational interpretation to Horn clauses, and it is close to lambda Prolog in spirit. We show some examples of programs written in Elf, such as theorem provers, type checkers and type inference algorithms, and evaluators for languages whose operational semantics is specified in LF. We close with some speculation on higher-order extensions (to the Calculus of Constructions) and some thoughts about module systems for this language, currently under investigation in joint work with Robert Harper. We also plan to give a demonstration of Elf, the latest version of which is written in ML.

- **Randy Pollack.** Implicit Syntax.

  Abstract. A proof checking system may support syntax that is more convenient for users than its 'official' language. For example LEGO has algorithms to infer some polymorphic instantiations (e.g. "pair 2 true" instead of "pair nat bool 2 true") and universe levels (e.g "Type" instead of "Type(4)"). Users need to understand such features, but do not want to know the algorithms for computing them. In this talk I will explain these two features much more simply by non-deterministic operational semantics for "translating" implicit syntax to a more explicit form. The translations are sound and complete for the underlying type theory, and the algorithms (which I will not talk about) are sound (not necessarily complete) for the translations.

Lunch 12:30 pm

Session 11. 2 pm Chair Bengt Nordström

- **Dale Miller.** An extension to ML to handle bound variables in data structures.

  Abstract. Most conventional programming languages have direct methods for representing first-order terms (say, via concrete datatypes in ML). If it is necessary to represent lambda-terms, formulas, types, or proofs (structures requiring the representation of bound variables), such structures must first be mapped into first-order terms, and then a significant number of auxiliary procedures must be implemented to manage bound variable names, check for free occurrences, do substitution, test for equality modulo alpha-conversion, etc. We shall show how the applicative core of the ML programming language can be enhanced so that lambda-terms can be represented more directly and so that the enhanced language, called ML_λ, provides a more elegant method of manipulating bound variables within data structures. In fact, the names of bound variables will not be accessible to the ML_λ programmer. This extension to ML involves the following: introduction of the new type constructor $\alpha \Rightarrow \beta$ for the type of lambda-terms formed by abstracting a parameter of type α out of a term of type β; a very restricted and simple form of higher-order pattern matching; a method for extending (at runtime) a given data structure with a new constructor; and, a method for extending (at runtime) function definitions to handle such new constructors. We shall give several examples of ML_λ programs and briefbly describe a prototype interpreter for it.

- **Amy Felty.** Encoding a dependent type λ-Calculus in a Logic Programming Language.

  Abstract. Various forms of typed λ-calculi have been proposed as specification languages for representing wide varieties of object logics. The *logical framework*, LF, is an example of such a dependent-type λ-calculus. A small subset of intuitionistic logic with quantification over simply typed λ-calculus has also been proposed as a framework for specifying general logics. The logic of *hereditary Harrop* formulas with quantification at all non-predicate types, denoted hh^ω, is such a meta-logic that has been implemented in both the Isabelle theorem prover and the λProlog logic programming language. In this talk, we show how LF can be encoded into hh^ω in a direct and natural way by mapping LF typing judgments in "$\beta\eta$-long normal" form into propositions in the logic of hh^ω. This translation establishes a strong connection between these two languages: the order of quantification in an LF signature is exactly the order of a set of hh^ω formulas, and the proofs in one system correspond directly to proofs in the other system.

- **Tobias Nipkow.** A Critical Pair Lemma for Higher-Order Rewrite Systems and its Application to Lambda-Star.

  Abstract. We consider rewrite systems over simply typed lambda-terms where the left-hand sides obey the following restriction: in their beta-normal form, any occurrence of a free variable x must be of the form $x(y_1 ... y_n)$, where the y_i are distinct bound variables. Calling such terms "templates" we obtain two important results:
  - Unification of templates is decidable and most general unifiers exist. This result is due to Dale Miller.
  - The critical pair lemma can be extended from first-order to such restricted higher-order rewrite systems.
  We can use this framework to formalize the reduction relation in various typed lambda-calculi and obtain confluence results by simple critical pair considerations. This is demonstrated for a particular system, lambda-star, which combines product and function types by imposing certain equalities between them, namely all the isomorphisms in a CCC minus commutativity of products. A potential application of this system to the unification problem for the typed lambda-calculus with ML-style polymorphism is sketched.

Tea Break

Session 12. 4:30 pm Chair Tobias Nipkow

- **Philippe de Groote.** Nederpelt's Calculus extended with a notion of Context as a Logical Framework.

  Abstract. This talk introduces the static part of a language intended to express software development. This language, which is called DEVA, has been under development during the ToolUse Esprit-project. Its static part amounts to an extended version of Nederpelt's Calculus. The extensions have been introduced in order to support the notion of mathematical definition of constants and to internalize the notion of theory. It results in a calculus which manipulates two kinds of objects : texts which correspond to lambda-expressions, and contexts which are sequences of variable declarations, constant definitions, or context abbreviations. The goal of the talk is to show how this extended calculus can be used as a logical framework.

- **Lincoln Wallen.** A reformulation of Herbrand's theorem.

  Abstract. We reformulate Herbrand's theorem using Gentzen methods to reveal the role of "permutability theorems" (in the sense of Kleene and Curry) relating to the conditions under which derivations are the same up to permutation of inferences. The key to resolution methods of proof-search is shown to be a full permutability theorem, enjoyed by (classical) propositional logic, but not by the predicate calculus. The occurs-check of unification is the means by which the combinatorial advantages of full permutability are feasibly extended to the predicate calculus. The reformulated Herbrand theorem—called a matrix theorem—unlike Herbrand's theorem itself extends to intuitionistic and (normal) modal systems (propositional and first-order versions). With it comes the combinatorics of resolution and an insight into the complexity of decision problems in terms of bounds on contraction. We sketch in what sense the matrix theorems give rise to Herbrand theorems for propositional logics.

7:30pm. Keynote address. Simon Bensasson. The future of Esprit's Basic Research Actions.

8pm Workshop Banquet

Friday, May 11th

BRA review. 9 am

Sites presentations

- **Gordon Plotkin.** General presentation of the BRA.
- **Cambridge site presentation**
- **Edinburgh site presentation**
- **Göteborg site presentation**
- **INRIA Rocquencourt site presentation**
- **INRIA Sophia-Antipolis site presentation**
- **Manchester site presentation**
- **Oxford site presentation**
- **Paris site presentation**
- **Turin site presentation**

Coffee Break

11am. Discussion session lead by the reviewers

Lunch 12:30 pm

2:30 pm. Review conclusion

3:00pm End of the workshop

Participants

Distinguished Guests

N. G. de Bruijn	(Eindhoven University)	[wsdwnb@win.tue.nl]
Robert Constable	(Cornell University, Ithaca)	[rc@gvax.cs.cornell.edu]
Jean Gallier	(U. of Pennsylvania, Philadelphia)	[Jean@central.cis.upenn.edu]
Philippe de Groote	(Université de Louvain la Neuve)	[drac@info.ucl.ac.be]
Susumu Hayashi	(Ryukoku University)	[hayashi@rins.ryukoku.ac.jp]
Robert Harper	(Carnegie Mellon U., Pittsburgh)	[rwh@PROOF.ERGO.CS.CMU.EDU]
Leen Helmink	(Philips Research, Eindhoven)	[helmink@neumann.prl.philips.nl]
Matt Kaufmann	(CLINC, Austin Texas)	[kaufmann@CLI.COM]
Dale Miller	(U. of Pennsylvania, Philadelphia)	[dale@linc.cis.upenn.edu]
Frank Pfenning	(Carnegie Mellon U., Pittsburgh)	[fp@CS.CMU.EDU]

Participants from Cambridge

Tobias Nipkow	[tnn@computer-lab.cambridge.ac.uk]
Martin Coen	[mc@lfcs.edinburgh.ac.uk]

Participants from Edinburgh

Alan Bundy	[bundy@aipna.edinburgh.ac.uk]
Gordon D. Plotkin	[GDP@ecsvax.ed.ac.uk]
Rod Burstall	[rb@cstvax.ed.ac.uk]
Randy Pollack	[rap@lfcs.ed.ac.uk]
Anne Salvesen	[abs@lfcs.edinburgh.ac.uk]
Claire Jones	[ccmj@lfcs.edinburgh.ac.uk]
James McKinna	[jhm@lfcs.edinburgh.ac.uk]
John Power	[ajp@lfcs.edinburgh.ac.uk]
David Pym	[dpym@lfcs.edinburgh.ac.uk]
George Cleland	[glc@lfcs.edinburgh.ac.uk]
Philippa Gardner	[pag@lfcs.edinburgh.ac.uk]
Alan Smaill	[smaill@aipna.edinburgh.ac.uk]

Participants from Göteborg

Bengt Nordström	[bengt@cs.chalmers.se]
Lennart Augustsson	[augustss@cs.chalmers.se]
Peter Dybjer	[peterd@cs.chalmers.se]
Lars Hallnäs	[lars@cs.chalmers.se]
Lena Magnusson	[lena@cs.chalmers.se]
Herbert Sander	[herbert@cs.chalmers.se]
Jan Smith	[smith@cs.chalmers.se]
Per Martin-Löf	

Participants from INRIA-Rocquencourt

Gérard Huet	[huet@margaux.inria.fr]
Thierry Coquand	[coquand@inria.inria.fr]
Christine Paulin	[cpaulin@lip.ens-lyon.fr]
Gilles Dowek	[dowek@margaux.inria.fr]
Benjamin Werner	[werner@margaux.inria.fr]
Andrea Asperti	[asperti@margaux.inria.fr]
Yves Lafont	[lafont@dmi.ens.fr]
Gabriel Stolzenberg	[gabe%zariski@harvard.harvard.edu]
Hugo Herbelin	[herbelin@margaux.inria.fr]

Participants from INRIA-Sophia-Antipolis

Gérard Boudol	[gbo@cma.cma.fr]
Yves Bertot	[bertot@mirsa.inria.fr]
Jacques Chazarain	[jmch@cerisi.cerisi.fr]
Joëlle Despeyroux	[jd@mirsa.inria.fr]
Thierry Despeyroux	[td@mirsa.inria.fr]
Amy Felty	[felty@mirsa.inria.fr]
Laurent Hascoët	[llh@mirsa.inria.fr]
André Hirschowitz	[ah@cerisi.cerisi.Fr]
Gilles Kahn	[kahn@mirsa.inria.fr]
Laurent Théry	[thery@mirsa.inria.fr]

Participants from Manchester

Peter Aczel	[petera@cs.man.ac.uk]
Nax Mendler	[mendler@cs.man.ac.uk]
Christian Jelstrup	
David Carlisle	[carlisle@cs.man.ac.uk]

Participant from Oxford

Lincoln Wallen	[Lincoln.Wallen@prg.oxford.ac.uk]

Participants from Paris

Michel Parigot	[pelz@sun8.lri.fr]
Jean-Yves Girard	[girard@margaux.inria.fr]
Gilles Amiot	
Vincent Danos	
Pascal Manoury	
Christophe Raffalli	
Paul Rozière	
Marianne Simonot	

Participants from Turin

Stefano Berardi	[stefano@ITOINFO.BITNET]
Mario Coppo	[coppo@ITOINFO.BITNET]
Paola Giannini	[giannini@ITOINFO.BITNET]
Furio Honsell	[HONSELL@UDUNIV.cineca.it]

Reviewers

Simon Bensasson	(E.E.C.)	[sben@dg13.cec.be]
Peter Schröder-Heister	(Universität Tübingen)	[PISCHROE@DKNKURZ1.bitnet]
Pierre Wolper	(Université de Liège)	[U502802@BLIULG11.bitnet]

Observers

Herman Geuvers	(Catholic University Nijmegen)	[herman@cs.kun.nl]
Ewan Klein	(University of Edinburgh)	[klein@cogsci.ed.ac.uk]
Mei Chee Shum	(University of Edinburgh)	

The Logical Theory of Constructions: A formal framework and its implementation

Peter Aczel and David P. Carlisle

Computer Science Department
Manchester University
Manchester, M13 9PL

Abstract

In this report we describe the implementation of a version of the LTC (Logical Theory of Constructions) framework in the Generic Theorem Prover *Isabelle*. An earlier version of the framework was described by Aczel and Mendler in [2].

1 Introduction

In [2] the notion of an open ended framework of deductive interpreted languages is formulated, and in particular an example is given of a hierarchy of languages L_i in the *LTC framework*. This paper is in two parts. In the first part we review this hierarchy of languages and then discuss some issues concerning the framework, which lead to another hierarchy of languages, LTC_0, LTC_1, LTC_ω. The second part of the paper documents the implementation of this last hierarchy in the generic theorem prover, Isabelle, developed by Larry Paulson at Cambridge [4, 5]. We also describe work in progress on verifying, in Isabelle, the interpretations of the type theories TT_0, TT_1 and TT_ω, that are presented in [6], in the corresponding languages of the new LTC hierarchy.

Part One: The Framework

2 The LTC framework and the languages L_i.

The increasing hierarchy of four languages L_i for $i = 1, \ldots, 4$ was originally defined in [2]. In keeping with the notion of framework presented there each language L_i has the form

$$(\sigma_i, T_i, I_i)$$

where σ_i is a signature of the framework, T_i is a theory for the signature and I_i is an interpretation of the theory with respect to the signature. Here we shall give a description of the basic framework and then go on to describe the languages L_i, ignoring for the most part in each case the, not to be implemented, interpretation component I_i of the language.

[1]This work has been partially supported by both SERC grant No. GR/E29861 and the ESPRIT Basic Research Action 'Logical Frameworks'.

It has been prepared for the first annual workshop of the Esprit Basic Research Action on Logical Frameworks, Antibes, May 1990.

2.1 The basic LTC framework

As presented in [2], the signatures of the LTC frameworks are taken over from the theory of arities as used by Per Martin-Lof in [7] and the Gothenberg group in [8]. This is essentially the typed lambda calculus, except that types are instead called arities and the notation for abstraction and application avoids any explicit symbols such as λ for abstraction or *app* for application. Each arity has the form

$$(\alpha_1 \ldots \alpha_n)$$

where $\alpha_1, \ldots, \alpha_n$ are arities and in case $n = 0$ we get the ground arity $0 = ()$. Such an arity is the arity of n-place function expressions having arguments of arities $\alpha_1, \ldots, \alpha_n$ and a value of the ground arity 0. A signature will be made up of a set of constant symbols, taken from a fixed sufficiently large set, each symbol being of some arity. The terms are built up in the usual way from these constants and the variables using abstraction and application. Terms that are interconvertible will be identified, so that in particular there is no notion of bound variable.

Each arity has a *level*, with the arity 0 having level 0 and for $n > 0$, the arity $(\alpha_1 \ldots \alpha_n)$ having level $l + 1$ where l is the maximum level of $\alpha_1 \ldots \alpha_n$. The arities of level 1 have the form

$$k = (\overbrace{0 \cdots 0}^{k})$$

for $k > 0$. For most purposes only symbols having an arity of level ≤ 2 are needed. The languages L_i only have such symbols. But just occasionally a symbol having an arity of level 3 seems to be useful and they have been used in the hierarchy of languages starting with LTC_0. It might be worthwhile to attempt to impose the restriction on the symbols of the LTC framework that their arity should always have level ≤ 2. The restriction comes down to only allowing individual variables to occur bound. It seems that it should be possible to meet this restriction by employing a few tricks, but we have not done so.

Every LTC signature will include a standard signature σ_0 for intuitionistic first order logic having the logical symbols $\perp$ of arity 0, $=, \supset, \wedge, \vee$ of arity 2 and the first order quantifiers $\forall, \exists$ of arity (1). In addition to the first order quantifiers we will also need in LTC_0 the quantifiers $\forall^1, \exists^1$ of arity ((1)) that quantify 1-place function variables of level 1. Every non-logical symbol will either be a function symbol, to be used in the formation of individual terms, or else a predicate symbol to be used in the formation of atomic formulae. This information needs to be given as part of the signature. With this information the individual terms and the formulae of the language can be inductively generated in more or less the standard way as in the predicate calculus.

Each theory for a given signature will include a standard system T_0 of schematic axioms and rules for the intuitionistic predicate calculus with equality. In addition to the usual rules for equality there are special rules for equality that express that the function symbols of the signature should denote injective functions with pairwise disjoint ranges. There are two kinds of special equality rule.

$$(SE1) \qquad \frac{c(f_1, \ldots, f_n) = c(g_1, \ldots, g_n)}{f_i(a_1, \ldots, a_{k_i}) = g_i(a_1, \ldots, a_{k_i})}$$

$$(SE2) \qquad \frac{c_1(f_{11}, \ldots, f_{1n_1}) = c_2(f_{21}, \ldots, f_{2n_2})}{\perp}$$

where in $(SE1)$ if c is an n-place function symbol of arity $(\alpha_1 \cdots \alpha_n)$then $1 \leq i \leq n$, f_j, g_j are terms of arity α_j for $1 \leq j \leq n$ and $a_1, \ldots a_{k_i}$ are terms of the appropriate arity, and in $(SE2)$ c_1, c_2 are distinct function symbols of arities $(\alpha_{11} \cdots \alpha_{1n_1})$ and $(\alpha_{21} \cdots \alpha_{2n_2})$ respectively and f_{1j} is a term of arity α_{1j} for $1 \leq j \leq n$ and f_{2j} is a term of arity α_{2j} for $1 \leq j \leq n_2$.

The motivation for having these special equality rules is that they are correct for the *LTC* interpretations, which are always certain kinds of term models; i.e. the first order quantifiers always range over sets of elements that are something like closed terms, but possibly in a larger signature.

2.2 The language $L_1 = (\sigma_1, T_1, -)$.

This language is a first order version of the Peano axioms for the natural numbers. The constructors of σ_1 consist of the logical symbols that are in σ_0 and in addition the symbols $0, S, N$. Here $0, S$ are function symbols of arities $0, 1$ respectively and N is a predicate symbol of arity 1. Two of the Peano axioms are already captured by the special equality rules for $0, S$.

$$\frac{S(a) = S(b)}{a = b} \qquad\qquad \frac{S(a) = 0}{\perp}$$

The remaining three Peano axioms need to be added explicitly to get the theory T_1.

$$\frac{}{N(0)} \qquad\qquad \frac{N(a)}{N(S(a))}$$

$$\frac{\phi(0) \qquad \forall x(\phi(x) \supset \phi(S(x)))}{N(a) \supset \phi(a)}$$

The language L_1 is still very weak as there is as yet no way to express addition and multiplication of natural numbers.

2.3 The language $L_2 = (\sigma_2, T_2, -)$.

This language extends L_1 by incorporating a simple functional programming notation with an explicit operational semantics. The *values* of the programming language are the individual terms of L_2 that are in canonical form; i.e. those individual terms whose outermost function symbol is canonical. Each function symbol of L_2 is stipulated to be canonical or non-canonical. The canonical constructors are $0, S$ from L_1, λ of arity (1) (lambda abstraction), $< -, - >$ of arity 2 (pairing) and inl, inr of arity 1 (left and right injections into a disjoint union). The control constructs of the programming language are the non-canonical function symbols. These are *Ap* of arity 2, *Spread* of arity (02), *Decide* of arity (011) and *Ind* of arity (001). The operational semantics for lazy evaluation of the expressions of the programming language is captured by using a predicate symbol $\rightsquigarrow$ of arity 2. This is intended to express the relation of evaluation to canonical form. The theory T_2 is obtained from T_1 by adding the appropriate evaluation rules for the control constructs.

So the language L_2 has the signature σ_2 obtained from L_1 by adding the function symbols $\lambda, < -, - >, inl, inr, Ap, Spread, Decide, Ind$ and the predicate symbol $\rightsquigarrow$. The evaluation rules are

$$(\rightsquigarrow 1) \qquad\qquad \frac{}{c \rightsquigarrow c}$$

if c is in canonical form,

$$(\rightsquigarrow 2) \qquad \frac{a \rightsquigarrow c \qquad d \rightsquigarrow e}{k(a) \rightsquigarrow e}$$

provided that (c, d, k) is a *computation triple*; i.e. one of the triples whose forms are listed in the following table. Note that the first column of the table lists all the canonical forms.

The Computation Triples of L_2.

c	d	k
$\lambda(f)$	$f(b)$	$(x)Ap(x,b)$
$< b_1, b_2 >$	$h(b_1, b_2)$	$(x)Spread(x,h)$
$inl(b)$	$f(b)$	$(x)Decide(x,f,g)$
$inr(b)$	$g(b)$	$(x)Decide(x,f,g)$
0	b	$(x)Ind(x,b,h)$
$S(c)$	$h(c, Ind(c,b,h))$	$(x)Ind(x,b,h)$

In the interpretation of L_2 the evaluation relation is inductively defined using the above two rules as the clause schemes of the inductive definition. It follows that the following two rules are also correct.

$$(\rightsquigarrow 3) \qquad \frac{a \rightsquigarrow e_1 \qquad a \rightsquigarrow e_2}{e_1 = e_2}$$

$$(\rightsquigarrow 4) \qquad \frac{a \rightsquigarrow c \qquad k(a) \rightsquigarrow e}{d \rightsquigarrow e}$$

for each computation triple (c, d, k). Although these two rules are not in L_2 we shall have them in the language LTC_0.

2.4 The language $L_3 = (\sigma_3, T_3, -)$.

The language L_3 can be viewed as a metalanguage for L_2, as it involves an internal representation of the semantics of L_2. To do this the signature σ_3 includes two new unary predicate symbols P_1 and T of arity 1. Also σ_3 includes a new canonical function symbol corresponding to each of the logical symbols and predicates of L_2. The arities of the new symbols are the same as the arities of the symbols of L_2 that they correspond to. These new function symbols will be indicated by placing a dot over the symbol they correspond to. The new rules of T_3 express in a fairly straightforward way a semantics for L_2, when the predicate symbol P_1 is used to internally represent the *propositions* of L_2 and the predicate symbol T is used to represent the *true* propositions of L_2. For each dotted symbol there is a *proposition formation rule* and also a rule giving the *truth conditions* for a proposition formed using the symbol. We can present the new rules of T_3 in a systematic way by associating with each individual term c of L_3 formulae Φ^1_c and Ψ_c as detailed below. The new rules for c are:-

$$\frac{e \leadsto c \quad \Phi^1_c}{P_1(e)} \qquad\qquad \frac{e \leadsto c \quad \Phi^1_c}{T(e) \equiv \Psi_c}$$

Here we use $\phi \equiv \psi$ to abbreviate the formula $(\phi \supset \psi) \wedge (\psi \supset \phi)$. The formulae Φ^1_c and Ψ_c are specified in the following table, where we use $\top$ to abbreviate $(\bot \supset \bot)$.

c	Φ^1_c	Ψ_c
$(a \dot{=} b)$	$\top$	$(a = b)$
$\dot{N}(a)$	$\top$	$N(a)$
$(a \dot{\leadsto} b)$	$\top$	$(a \leadsto b)$
$\dot{\bot}$	$\top$	$\bot$
$(a \dot{\wedge} b)$	$P_1(a) \wedge P_1(b)$	$T(a) \wedge T(b)$
$(a \dot{\vee} b)$	$P_1(a) \wedge P_1(b)$	$T(a) \vee T(b)$
$(a \dot{\supset} b)$	$P_1(a) \wedge (T(a) \supset P_1(b))$	$T(a) \supset T(b)$
$\dot{\forall}(f)$	$\forall x P_1(f(x))$	$\forall x T(f(x))$
$\dot{\exists}(f)$	$\forall x P_1(f(x))$	$\exists x T(f(x))$

2.5 The language $L_4 = (\sigma_4, T_4, -)$.

The language L_3 was obtained by a process of reflection on the language L_2. This process can be repeated to obtain a metalanguage for L_3, a metalanguage for that language and so on. The language L_4 is a language incorporating the result of repeating the reflection process any finite number of times. So there is now a hierarchy of predicate symbols P_i for $i = 1, 2, \ldots$ with P_{i+1} being used to internally represent the propositions of the sublanguage of L_4 that only involves the predicate symbols $P_1, \ldots, P_i$. There need to be dotted function symbols $\dot{P}_i$ corresponding to these predicate symbols. The rules of T_4 are obtained by adding to the rules of T_2 the following rules for $i = 1, 2, \ldots$

$$\frac{e \leadsto c \quad \Phi^i_c}{P_i(e)} \qquad\qquad \frac{e \leadsto c \quad \Phi^i_c}{T(e) \equiv \Psi_c}$$

$$\frac{P_j(e)}{P_i(e)} \;\; (j < i)$$

Here Φ_c^i is defined like Φ_c^1 using P_i instead of P_1, except that there is the following additional row in the table defining the formulae Φ_c^i and Ψ_c.

c	Φ_c^i	Ψ_c
$\dot{P}_j(a)$ (with $j < i$)	$\top$	$P_j(a)$

3 Discussion

In this section we focus on some issues that are relevent to an understanding of the *LTC* framework as presently conceived. A suitable notion of term model is basic to the interpretations of the *LTC* languages and this is discussed in 3.2. Because the defined and primitive symbols are treated differently in term models we start in 3.1 by reviewing the familiar notion of a *definitional extension*. We then formulate a generalisation, the notion of a *recursion extension*. Finally we look at the distinction between *numerals* and *natural numbers* that exists in the context of a lazy evaluation relation and also formulate the additional notion of full evaluation that seems to be needed for the natural numbers.

3.1 Definitional Extensions

The notion of a *definitional extension* of a theory is a familiar one that makes sense for many logics. The idea is to extend a theory by adding to it new defined symbols with their defining axioms. The new theory should be an inessential extension in the sense that the defined symbols can always be systematically eliminated. We look more closely at definitional extensions in the case of standard first order logic.

We assume given a first order theory consisting of a first order signature of function and predicate symbols. The signature of the theory is extended with some new function and predicate symbols and with each new symbol is associated a definition which is formulated as an equality for a function symbol and is formulated as a logical equivalence for a predicate symbol. These definitions are added as new axioms to the old theory to obtain a new theory. The new theory is then called a definitional extension of the old one. In the case of a defined n-place function symbol F the definition has the form

$$F(x_1, \ldots, x_n) = e$$

where $x_1, \ldots, x_n$ is a non-repeating list of individual variables and e is an individual term of the old theory whose free variables are among $x_1, \ldots, x_n$. If F is a predicate symbol then e should be a formula

whose free variables are among $x_1, \ldots, x_n$ and equality should be replaced by logical equivalence.

Definitional extensions are deductively inessential in the sense that all defined symbols can in principle be eliminated from terms and formulae in a systematic way using the definitions. It follows that a definitional extension is conservative over the original theory; i.e. any theorem of the new theory that is expressed in the old signature will already be a theorem of the old theory.

Definitional extensions are also semantically inessential in the sense that any model of the old theory will have a unique enlargement to the new theory. The enlargement is defined by interpreting each new symbol in the model in the obvious way using the interpretation of the right hand side of its definition.

The *LTC* framework is based on a nonstandard version of first order logic in that the function and predicate symbols are allowed to be variable binding. It is a straightforward matter to work out the notion of a definitional extension where definitions of variable binding functions are allowed. As in the standard case definitional extensions are still deductively and semantically inessential.

In general the signature of a theory will have two kinds of symbol, those that are primitive and those that have been defined in terms of the primitive ones. For certain purposes, such as the notion of term model below, it can be important to keep the two kinds of symbol distinct. The primitive symbols will be called *constructors* and the remaining symbols the *defined symbols*.

3.2 Term Models

The notion of a term model for a theory makes sense for many logics and is often useful. The general idea is that the elements of a term model are terms or something like terms. In the LTC framework a notion of term model will play an essential role, as only term models of a particular kind will be allowed as interpretations. Below we consider something like the particular notion of a term model that will be used in the LTC framework in the more familiar and simpler context of first order logic.

Given a first order signature, with no defined symbols, the underlying set for a term model for that signature consists of the set of closed terms of some signature extending the given one with additional function symbols. Each function symbol of the given signature is interpreted in the standard way using the symbol itself. The predicate symbols of the given signature are free to be interpreted in any way over the underlying set. So a term model is specified by giving the additional function symbols that may be used in forming elements and then interpreting the predicates over the resulting set of terms.

The notion of a term model still makes sense in the presence of variable binding symbols. The set A_0 of terms that make up the elements of the term model now consist of the closed terms; i.e. terms having no free variables, up to α-convertibility. The set A_n of n-place functions of level 1 of a term model will consist of n-place functions f on A_0, each such f being given by a closed n-place abstract

$$(x_1, \ldots, x_n)a$$

so that

$$f(a_1, \ldots, a_n) = a[a_1, \ldots, a_n/x_1, \ldots, x_n]$$

for all $a_1, \ldots, a_n \in A_0$.

In the case of a signature with defined symbols as well as constructors it is important to the notion of a term model that only the constructors are used in forming the elements of the model. The correctness of the special equality axioms in these term models should be clear. Of course the special equality axioms are only for the constructors and not for the defined symbols.

3.3 Recursion Extensions and their term models

Here we wish to consider a generalisation of the notion of a definitional extension, which we will use in the language LTC_0. In the LTC framework there is a clear separation between those aspects that are to do with definitions and the resulting notion of definitional equality and those aspects that are to do with the computation triples and the resulting evaluation relation. These aspects are conflated in the type free lambda calculus and in the original notion of Frege structure,[1], that was used as the first idea for the interpretations of something like the LTC framework,[9]. By our use of explicit evaluation relations over term models we keep these aspects separate. Our idea here is to compromise between the two approaches by allowing definitions to be recursive and only leaving discrimination/selection to be represented in the computation triples. Take the example of the non-canonical constructor Ind of the language L_2. It is used to capture primitive recursion on the natural numbers, a computational notion. We can analyse primitive recursion into the two separate components of recursion and discrimination/selection and move the recursion component to the definitional aspect, keeping only the discrimination/selection features of computation to be captured with the computation triples. To do this we now take Ind to be a defined symbol with recursive defining equation

$$Ind(c,b,h) = DecideNat(c,b,(x)h(x,Ind(x,b,h))).$$

Here $DecideNat$ is to be a new non-canonical constructor used in the following computation triples.

c	d	k
0	b	$(x)DecideNat(x,b,g)$
$S(c)$	$g(c)$	$(x)DecideNat(x,b,g)$

These replace the old computation triples for the constructors $0, S$ that used Ind. The constructor $DecideNat$ is purely a discriminator/selector as are the other non-canonical constructors of L_2,Ap, $Spread$ and $Decide$.

The idea for this kind of separation of recursion from discrimination/selection came from a reading of [3]

although they do not distinguish between equality and evaluation in the way we want to.

Let us now describe the general notion of a *recursion extension* of a theory. This notion will generalise the notion of a definitional extension. Although no longer necessarily always an inessential extension it turns out that a recursion extension will still have a reasonable notion of term model. A recursion extension is a new theory obtained from an old one by adding new defined symbols to the signature and for each defined symbol, F, a defining equation

$$F(x_1,\ldots,x_n) = e$$

where e is an expression of the new theory (so that it may contain F itself) whose free variables are among $x_1,\ldots,x_n$. Here if F is a symbol of arity $(\alpha_1,\ldots,\alpha_n)$ then each x_i is a variable of arity α_i. The defining equation is taken as an axiom scheme of the new theory, so that all instances obtained by substituting for the variables will be axioms.

A key fact about a signature of constructors and defined symbols, where each defined symbol has a defining equation as above, is that the Church-Rosser property holds for the following reduction notion. Every β redex

$$((x_1, \ldots, x_n)a)(a_1, \ldots, a_n)$$

is a redex that contracts to

$$a[a_1, \ldots, a_n/x_1, \ldots, x_n].$$

But also, for every defining equation as above every instance determines a redex

$$F(a_1, \ldots, a_n)$$

that contracts to

$$e[a_1, \ldots, a_n/x_1, \ldots, x_n].$$

The details concerning this fact will be discussed elsewhere. What the fact implies for us is that there is a good notion of term model which will satisfy all the defining equations and also all the equality rules including the special ones. Roughly the set of individuals of the term model is to be defined by forming a quotient of the set of closed terms by the equivalence relation of provable equality between closed terms.

We have made one change to the table of computation triples. The triples involving Ind were replaced by those involving $DecideNat$. With this change the computation triples are formed in a pretty uniform way. Only the triple for the canonical constructor λ departs slightly from a very clearcut pattern. The canonical constructors form into subsets of related symbols. While λ and $<,>$ are on their own inl, inr are related and so are $0, S$. With each maximal set

$$\{c_1, \ldots, c_m\}$$

of related canonical constructors there is associated a non-canonical constructor Δ and computation triples given in the following table

c	d	k
$c_1(f_{11}, \ldots, f_{1n_1})$	$h_1(f_{11}, \ldots, f_{1n_1})$	$(x)\Delta(x, h_1, \ldots, h_m)$
$\vdots$	$\vdots$	$\vdots$
$c_m(f_{m1}, \ldots, f_{mn_1})$	$h_m(f_{m1}, \ldots, f_{mn_1})$	$(x)\Delta(x, h_1, \ldots, h_m)$

The three symbols that play the role of Δ for the three sets $\{<,>\}, \{inl, inr\}$ and $\{0, S\}$ are, of course, $Spread, Decide$ and $DecideNat$. To get a computation triple for $lambda$ to fit the pattern we need a non-canonical constructor Pa and the computation triple

c	d	k
$\lambda(f)$	$h(f)$	$(x)Pa(x, h)$

The symbol Ap can now be introduced as a defined symbol with definition

$$Ap(x, b) = Pa(x, (f)f(b)).$$

Note that Pa is a symbol of arity $(0(0))$, an arity of level 3.

3.4 Natural numbers and their full evaluation

The predicate N of the language L_1 expresses the property of being a *numeral* in unary notation; i.e. being one of $0, S(0), S(S(0)), \ldots$. This notion of numeral is inadequate for computational purposes in the *LTC* framework. We need a notion of *natural number* which will guarantee, for example, that $plus(a, b)$ will be a natural number whenever a, b are. Here *plus* is defined in terms of *Ind* so as to have the following evaluation rules

$$(plus1) \qquad \frac{a \leadsto c \qquad b \leadsto 0}{plus(a, b) \leadsto c}$$

$$(plus2) \qquad \frac{b \leadsto S(c)}{plus(a, b) \leadsto S(plus(a, c))}.$$

It will not do to define a natural number to be an object that evaluates to a numeral. This is because our notion of evaluation is lazy and objects are only evaluated to outermost canonical form. For example $plus(S(0), S(0))$ evaluates to $S(plus(S(0), 0))$ which is not a numeral. The correct definition is that a natural number is an object that is inductively generated by the rules $(Nat1)$ and $(Nat2)$ below. The rule $(Nat3)$ is the principle of mathematical induction for this notion of natural number.

$$(Nat1) \qquad \frac{a \leadsto 0}{Nat(a)}$$

$$(Nat2) \qquad \frac{a \leadsto S(b) \qquad Nat(b)}{Nat(a)}$$

$$(Nat3) \qquad \frac{\forall x(x \leadsto 0 \supset \phi(x)) \qquad \forall x, y(x \leadsto S(y) \;\&\; \phi(y) \;\supset\; \phi(x))}{Nat(a) \supset \phi(a)}$$

Every natural number should have a full evaluation to a numeral. But it is not clear if this notion of full natural number evaluation can be expressed in L_2 even when Nat and its rules have been added. So we want a relation $\rightsquigarrow\!\!\!\rightsquigarrow$ to express this notion of full evaluation. Its rules are

$(\rightsquigarrow\!\!\!\rightsquigarrow 1)$

$$\frac{a \leadsto 0}{a \rightsquigarrow\!\!\!\rightsquigarrow 0}$$

$(\rightsquigarrow\!\!\!\rightsquigarrow 2)$

$$\frac{a \leadsto S(b) \qquad b \rightsquigarrow\!\!\!\rightsquigarrow c}{a \rightsquigarrow\!\!\!\rightsquigarrow S(c)}$$

$(\rightsquigarrow\!\!\!\rightsquigarrow 3)$

$$\frac{\forall x(x \leadsto 0 \supset \phi(x, 0)) \qquad \forall x, y, z(x \leadsto S(y) \;\&\; \phi(y, z) \;\supset\; \phi(x, S(z)))}{(a \rightsquigarrow\!\!\!\rightsquigarrow b) \supset \phi(a, b)}$$

The first two rules give the clauses for the inductive definition of the full evaluation relation and the final rule expresses proof by induction following the inductive definition. Using these rules we can derive the rules for N and Nat when these are defined by

$$N(x) \equiv x \leadsto\!\!\leadsto x,$$

$$Nat(x) \equiv \exists n(x \leadsto\!\!\leadsto n).$$

We can also define the relation $EqNat$ of natural number equality by

$$EqNat(x, y) \equiv \exists n(x \leadsto\!\!\leadsto n \ \wedge \ y \leadsto\!\!\leadsto n).$$

This is used in defining the type of natural numbers. So we shall take the rules for $\leadsto\!\!\leadsto$ as primitive in LTC_0 and use the above definitions of N and Nat.

4 The Languages LTC_0, LTC_1 and LTC_ω.

In this section we describe the three languages. Our description can be brief as these languages are only slightly different to the languages L_2, L_3 and L_4 of [2] and section 2, and the points of difference have been discussed in the previous section.

The LTC_0 theory uses a natural deduction style axiomatisation of intuitionistic logic with equality for the first order logical symbols $\perp, =, \supset, \wedge, \vee, \forall, \exists$ and the second order quantifiers $\forall^1, \exists^1$ of arity ((1)) that quantify 1-place functions from individuals to individuals. The non-logical symbols are made up of

Function Symbols

- Canonical constructors: $0, S, \lambda, <, >, inl, inr$
- Non-canonical constructors: $DecideNat, Pa, Spread, Decide$
- Defined function symbols: Ap, Ind

Predicate Symbols

- Primitive: $\leadsto, \leadsto\!\!\leadsto$
- Defined: N, Nat

The non-logical rules of LTC_0 consist of:

- The special equality axioms ($SE1$), ($SE2$) of 2.1 for the constructors.
- The evaluation rules ($\leadsto$ 1), ($\leadsto$ 2), ($\leadsto$ 3) and ($\leadsto$ 4) of 2.3 using the computation triples tabled there except that the computation triples involving Ind and Ap are replaced by those involving $DecideNat$ and Pa given in 3.3.
- The full evaluation rules ($\leadsto\!\!\leadsto$ 1), ($\leadsto\!\!\leadsto$ 2) and ($\leadsto\!\!\leadsto$ 3) of 3.4.
- The defining equations for the defined symbols Ind, Ap, N, Nat, as presented in 3.3 and 3.4.

The languages LTC_1 and LTC_ω are obtained from LTC_0 using exactly the same kind of internal reflection and its iteration that are used in the formation of L_3 and L_4 from L_2. Instead of the constructor $\dot{N}$ of L_3 there is now a constructor $\dot{\leadsto}$. Also there are new constructors $\dot{\forall}^1, \dot{\exists}^1$ reflecting the function quantifiers of LTC_0.

Part Two: The Implementation

5 Isabelle

By a 'Generic Logic Proof Development System', we mean a system in which one expresses an *Object Logic* under consideration in a *Meta Logic*. The system then furnishes a proof development system for that logic i.e. mechanisms for deriving and storing rules and theorems of the object logic, matching rules to goals, and a programmable *Meta Language* in which one may write Tactics or Proof Strategies. In fact Isabelle uses Huet's higher order unification algorithm rather than matching. There are now quite a few such systems, but Isabelle seems to be particularly well suited to the notion of a *Framework* developed in [2], which has similarities to the Isabelle notion of a *Theory* and its *extensions*.

The Meta Logic of Isabelle is a version of higher order logic. The terms are those of the typed lambda calculus. Initially the only ground type is *prop* (`Aprop`). When implementing an object logic new ground types and constants may be added. There are also function types $\sigma \to \tau$ where σ and τ are types.

Isabelle implements higher order unification (i.e. unification up to α and β reduction) as the main mechanism for constructing proofs. Note that as unification rather than matching is used schematic variables appearing in a goal may be instantiated as well as those appearing in the rules. Usually one does not want schematic variables in the main goal as this may result in a proof of only a special case of the goal. Unification is mainly used in the instantiation of variables that have been introduced into subgoals produced by previous proof steps. The framework has been implemented as a series of *extensions* to the standard Isabelle theory FOL which implements a natural deduction style intuitionistic First Order Logic with equality.

6 The Logical Theory of Constructions

In this section we give an outline of our Isabelle implementation of the languages LTC_0, LTC_1 and LTC_ω.

6.1 `LTC0`

This is an implementation of LTC_0.

The Isabelle notation for specifying the types (arities) of constructors is different to that used in section 2, however the translation is quite simple. We introduce two ground types *term* and *form*, then if the arities $\alpha_1, \ldots, \alpha_n$ correspond to the isabelle types $t_1, \ldots, t_n$, a function of arity $(\alpha_1 \ldots \alpha_n)$ has type $t_1 \to \ldots \to t_n \to \mathit{term}$, and a predicate of arity $(\alpha_1 \ldots \alpha_n)$ has type $t_1 \to \ldots \to t_n \to \mathit{form}$. Thus for example the function symbol *Decide* of arity (011) corresponds to a function of type $\mathit{term} \to (\mathit{term} \to \mathit{term}) \to (\mathit{term} \to \mathit{term}) \to \mathit{term}$. The Isabelle declaration of the types of the constants of `LTC0` is shown below.

```
 Infixr("~>", [Aterm,Aterm]--->Aform, 60),
 Infixr("~~>", [Aterm,Aterm]--->Aform, 60),
 Delimfix("<_,_>", [Aterm,Aterm]--->Aterm, "Pair"),
 Delimfix("0", Aterm, "0"),
 Delimfix("S", Aterm-->Aterm, "S"),

. . .

 (["Ind"],  [Aterm, Aterm, [Aterm,Aterm] ---> Aterm] ---> Aterm),
 (["Spread"], [Aterm, [Aterm,Aterm] ---> Aterm] ---> Aterm),
 (["Decide"], [Aterm,(Aterm --> Aterm),(Aterm --> Aterm)] ---> Aterm),

 (["Lambda"], (Aterm-->Aterm) --> Aterm),
 (["Ap"], [Aterm,Aterm]--->Aterm),
 (["Inl","Inr"], Aterm-->Aterm),
 (["Pa"], [Aterm,(Aterm-->Aterm)-->Aterm]--->Aterm),
 (["DecideNat"], [Aterm,Aterm,Aterm-->Aterm]--->Aterm),

 (["N"], Aterm-->Aform),
 (["Nat"], Aterm-->Aform)
```

Here `-->`, `Aterm` and `Aform` correspond to $\to$, *term*, and *form* respectively, and $[T_1, \ldots, T_n]$ `--->` T is a shorthand notation for T_1`-->` $\cdots$ `-->`T_n`-->`T.

The first five constants are declared using Isabelle's new mixfix syntax. This is used to declare the concrete syntax for a constant, if this is different from the normal, prefix application. Thus `~>` and `~~>` are infix operators, and the notation `<a,b>` is equivalent to `Pair(a,b)`.

`0` and `S` are treated in a special way as the `LTC0` parser allows, for instance, either `#N3` or `S(S(S(0)))` as concrete syntax for the term $S(S(S(0)))$ of LTC_0. The details of this *parse translation* and its associated *print translation* will be omitted.

There is one further mixfix declaration:

```
Mixfix("LAM _. _",[id_list,Aterm]--->Aterm," LAM",[],10),
```

This is somewhat special as it does not result in a constant being defined in the

abstract syntax for `LTC0`. The details of the parse translation will again be omitted, but this declaration allows `LAM x y. f(x,y)` as an alternative to `Lambda(%x.Lambda(%y.f(x,y)))`.

The other constants of `LTC0` require no special syntax, they are listed with their types.

In Isabelle there is no difference between the representations of rules and theorems. both are represented using the implication $\Rightarrow$ (`==>`) of Higher Order Logic. The rules are given in a textual form, together with a name. For instance we have the following rule.

```
"(a ~> Inr(b)) ==> (g(b) ~> e) ==> Decide(a,f,g) ~> e"
```

Note that in the latest version of Isabelle, the function mapping object-language truth values (*form*) to meta-language truth values (*prop*) is "invisible" (The parser applies the function `Trueprop` of type `Aform-->Aprop` when necessary). Because of this feature, one has to explicitly constrain terms to type *prop* if they appear in rules, by following the term with `$$prop`.

The above rule corresponds to the axiom

$$\frac{a \leadsto inr(b) \quad g(b) \leadsto e}{decide(a, f, g) \leadsto e}$$

of the language L_2, of [2]. In LTC_0, we wish to derive this rule from ($\leadsto$ 2) and the fact that $(inr(b), g(b), (x)Decide(x, f, g))$ is a computation triple. In order to express this fact in `LTC0`, we introduce a new constant `ALPHA` of type `[Aterm,Aterm,Aterm-->Aterm] ---> Aprop`. `ALPHA(c,d,k)` expresses at the meta level the fact that (c, d, k) is a computation triple.

Thus corresponding to ($\leadsto$ 1) and ($\leadsto$ 2) we have the following two axioms of `LTC0`

```
("A_canon", " ALPHA(c,d,k)$$prop ==>  c ~> c "),
("A_non_canon", "[|ALPHA(c,d,k)$$prop ; a~>c ; d~>e|]  ==>  k(a) ~> e"),
```

The computation triple above is implemented as:

```
("A_inl", " ALPHA(Inl(b) ,g(b) ,%x. Decide(x,g,h))$$prop "),
```

The special inequality rules of the LTC are particularly awkward to implement. $SE1$ is reasonably straight forward, for each constructor we need to add rules such as

```
("inj_pair_1", "Pair(a,b) = Pair(a',b') ==> a = a'" ),
("inj_pair_2", "Pair(a,b) = Pair(a',b') ==> b = b'" ),
```

One way to implement $SE2$ would be to add, for each pair of constructors, say S and $lambda$ an axiom of the form

```
  ("S_lam",     " S(a) = Lambda(f)  ==>  False " ),
```

There are 10 constructors in the language LTC_0 and so there would be 45 rules like the one above. In any extension of the theory, when adding the nth constructor, $n-1$ rules would have to be added. This is clearly not desirable, as even if the input of these axioms were automated, the proof tactics would have to search over an ever growing list of rules, just to prove an instance of the rule $SE2$.

Such considerations have lead us to introduce a new ground type, *token* into the meta theory, together with four new constructors, with the following type declarations:

```
    Delimfix("t0", Atoken, "t0"),
    Delimfix("tS", Atoken-->Atoken, "tS"),
 . . .
    (["Token"], Aterm --> Atoken),
    (["DiffToken"], [Atoken,Atoken]---> Aprop),
```

Parse translation functions allow us to write `#T3` for the token $tS(tS(tS(t0)))$.

$SE2$ may now be encoded as:

```
("token","[| a = b ; (DiffToken(Token(a),Token(b)))$$prop |] ==> False"),
```

Now, for each constructor, we only need to add one rule, such as

```
("E_S", "Token(S(b)) == #T6" ),
```

Which assigns a unique token to each constructor.

The other axioms related to tokens are quite straightforward:

```
("tSS", "(DiffToken(a,b))$$prop ==> (DiffToken(tS(a),tS(b)))$$prop" ),
("tSZ", "(DiffToken(tS(a),t0))$$prop" ),
("tZS", "(DiffToken(t0,tS(a)))$$prop" ),
```

The constants *Ind* and *Ap* are defined by meta level equalities as discused in section 3.3:

```
("ind_def", "Ind(c,b,h) == DecideNat(c,b,%x. h(x,Ind(x,b,h)))"),
("ap_def", "Ap(x,b) == Pa(x,%f. f(b))")
```

6.2 LTC1

`LTC1` is an Isabelle extension of `LTC0` corresponding to LTC_1. The main interest here concerns the introduction of a new meta-type, `Alevel`.

LTC_1 only has one predicate symbol P_1, to express a single level of internal propositions. LTC_ω has an infinite sequence of predicate symbols P_1, P_2, ... expressing an infinite hierarchy of levels of propositions. In the implementation `LTC1`, a constant `Prop` of type `Alevel --> Aterm --> Aform`, is introduced, rather than simply using a constant `P1` of type `Aterm --> Aform`. In `LTC1` the type `Alevel` has only one constructor, `lev_one`.

The reader might expect the rules of `LTC1` to refer to `Prop(lev_one,a)`, however this would mean that the extension `LTCW`, corresponding to LTC_ω, would have to re-implement the rules for higher levels. In fact, the rules implemented for `LTC1` all involve a meta-variable `lev` ranging over the type `Alevel`. This does have the drawback that in some circumstances schematic variables ranging over `Alevel` are introduced during a proof. In the usual situation, where one is attempting to prove a theorem by working backwards from the goal which has been entered by the user, these schematic variables ranging over `Alevel` will be instantiated to `lev_one` before the proof is finished. In `LTCW` the type `Alevel` is augmented with a unary constructor `lev_S` denoting the successor function on the natural numbers. The parsing and printing functions are defined to so that, for example `lev_S(lev_S(lev_one)` is treated as `#L3`.

The names of the 'dotted' term forming versions of the logical constructors are formed by prepending a `D` or `.` as appropriate to the name of the logical constructor, e.g. `DForall` and `.-->`.

Thus corresponding to the pair of rules

$$\frac{e \rightsquigarrow a \dot{\wedge} b \quad P_1(a) \qquad P_1(b)}{P_1(e)\ ,\ T(e) \wedge T(a) \supset T(b)}$$

of LTC_1, we have the following pair of rules in `LTC1`.

```
("Dconj_P",
   "[| e ~> (a .& b); Prop(lev)(a); Prop(lev)(b) |] ==> Prop(lev)(e)" ),
("Dconj_T",
   "[| e ~> (a .& b); Prop(lev)(a); Prop(lev)(b) |] ==>  \
\                                        T(e) <-> (T(a) & T(b))" ),
```

In the present implementation, these rules are given as axioms. It would be possible to introduce a new meta-level predicate, `BETA`, corresponding to the table of section 2.4, just as `ALPHA` coresponds to the table of section 2.3. These rules would then be derivable.

The rules for T are given in the same style as [2], but they are not well suited to Isabelle's proof style. The `LTC1` tactics use, instead, derived rules such as:

```
Dconj_T_elim
"[| T(e); e ~> (a .& b); Prop(lev)(a); Prop(lev)(b); \
\                                         (T(a) & T(b) ==> R)|]  ==> R"

Dconj_T_intr
"[| e ~> (a .& b); Prop(lev)(a); Prop(lev)(b); (T(a) & T(b)) |] ==> T(e)"
```

In [2] it is mentioned that in the theory for the language L_3, (or LTC_1) neither P_1 nor T can be internally defined, i.e. the following two theorems hold.

$$\neg\exists t.\, \forall x.\, P_1(Ap(t,x)) \wedge (T(Ap(t,x)) \Leftrightarrow T(x))$$

$$\neg\exists p.\, \forall x.\, P_1(Ap(p,x)) \wedge (T(Ap(p,x)) \Leftrightarrow P_1(x))$$

These may be viewed as analogues of Russell's paradox. The file russell.ML develops a series of lemmas culminating in the following theorems of `LTC1`

```
" ~(EX t. ALL x. Prop(#L1)(Ap(t,x)) & (T(Ap(t,x)) <-> T(x))) "
" ~(EX p. ALL x. Prop(#L1)(Ap(p,x)) & (T(Ap(p,x)) <-> Prop(#L1)(x))) "
```

The above theorems demonstrate the way that Isabelle allows the concrete syntax of the object logic implemented to be very close to the concrete syntax normally used for that logic. (An earlier version of Isabelle allowed the better syntax `P_1(a)` for $P_1(a)$, it seems hard to do this for all levels with the 1990 version of Isabelle). `<->` is defined by the Isabelle *definition* mechanism i.e. meta level equality, FOL has the rule

```
("iff_def",  "P<->Q == (P-->Q) & (Q-->P)"),
```

`EX`, `ALL` and `#` are all delimiters which form part of Isabelle's system of *notation*, thus they may be considered as part of definitions which are always unfolded by the parser, and folded by the printer. *Definitions* such as that for `<->` are not unfolded unless this is explicitly demanded, derived rules are proved, and then the defined constructor may be used in just the same way as the primitive ones. This ability to tailor the syntax to your requirements more than makes up for Isabelle's complete lack of a user interface. While the provision of some nicer fonts would improve the look of the output, these extra characters should be able to be *typed* in.

6.3 LTCW

`LTCW` is an Isabelle extension of `LTC1` corresponding to LTC_ω. The implementation of `Alevel` is described above. We introduce a new constant, `lev_S`, of type `Alevel-->Alevel`. The rules given in `LTC1` then immediately apply to all levels.

The expression forming version of P_i, $\dot{P}_i$, is treated in the same way as P_i . A constant DProp of type Alevel --> Aterm --> Aterm, is declared.

The only further complication arises from the implementation of the side condition $j < i$ on the rules

$$\frac{e \rightsquigarrow \dot{P}_j(a)}{P_i(e)} \quad \text{and} \quad \frac{P_j(e)}{P_i(e)}.$$

We introduce a new constant, lower_lev, of type Alevel-->Alevel-->Aprop. Note that Aprop is the type of meta-level truth values, not the object-level Aform. We then add a minimal set of axioms that allow us to deduce that one level is lower than another, namely:

```
("lower_S", "(lower_lev (j,i))$$prop ==> \
\             (lower_lev(lev_S(j) , lev_S(i)))$$prop"),
("lower_1", "(lower_lev(lev_one ,lev_S(i)))$$prop")
```

The remaining rules for LTC_ω may then be coded as:

```
("A_Dprop",  "ALPHA(DProp(l,a), g(l,a), %x.DecideDP(x,g))$$prop"),
("E_Dprop", "Token(DProp(l,a)) == #T19" ),

("DPj_Pi", "(lower_lev(j,i))$$prop ==> e ~> DProp(j)(a) ==> Prop(i)(e)"),
("DP_T", "e ~> DProp(j)(a) ==> T(e) <-> (Prop(j)(a))"),
("Pj_Pi", "(lower_lev(j,i))$$prop ==> Prop(j)(a) ==> Prop(i)(a)"),
```

The first rule is mainly use to express the fact that the constructors $\dot{P}_1$, $\dot{P}_2$ ... are canonical. The non canonical DecideDP is not used at present. the rule E_Dprop just assigns the token #T19 to Dprop as described earlier. Note that this will not allow us to deduce $\neg(\dot{P}_1(a) = \dot{P}_2(a))$. An axiom capturing $SE2$ for different levels is required (but not currently implemented).

7 Type Theory

One of the motivating ideas behind the Logical Theory of Constructions, was that it should provide a logical framework in which Martin-Löf style Type Theories could be defined. In [6], Nax Mendler gives a description of three type theories. The first without universes, the second with one universe, and the third with a countable hierarchy of universes. He also gives translations of the theories into LTC_0, LTC_1, LTC_ω respectively.

We are currently implementing these three theories in Isabelle. The idea is to define Isabelle theories TT0, TT1 and TTW as extensions of the theories described above. These extensions only consist of declarations of new constants, and rules which are all meta-level equalities giving definitions for the new constants in terms of the LTC. The axioms of the Type theories should then all be derivable. We have not yet finished deriving all these rules, but expect to do so in the near future.

7.1 TT0

TT0 corresponds to the theory TT_0 of [6]. It is built as an Isabelle extension of LTC0.

Types are interpreted as binary predicates satisfying certain restrictions, ie of type (in the LTC) *term* $\rightarrow$ *term* $\rightarrow$ *form*. As this occurs very often in the constant declarations below, the ML identifier `Atype` is bound to `Aterm-->Aterm--Aform`. `Atype` does not refer to a new ground type *type* of the meta-language. Using this abbreviation has the advantage that these files could easily be converted to define a primitive Isabelle theory (not an extension of `FOL/LTC`) in which `Aterm`, `Atype` and `Aform` are three ground types corresponding to terms types and judgments in the Type Theory.

The constants are

```
  Mixfix("<| _ ~= _ in _ |>", [Aterm,Aterm,Atype]--->Aform,"EqElem", ...
  Mixfix("<| _ ~= _ |>",        [Atype,Atype]--->Aform,"EqType", ...

  Mixfix("PROD _:_. _",    [SId,Atype,Atype]--->Atype, " PROD", [], 10),
  Mixfix("SUM _:_. _", [SId,Atype,Atype]--->Atype, " SUM", [], 10),
  Infixr("+", [Atype,Atype]--->Atype, 30)
 . . .
    (["NAT","void"],    Atype ),
    (["I"],             [Atype,Aterm,Aterm] ---> Atype ),
    (["true"], Aterm ),
    (["any"], Aterm-->Aterm  ),
    (["Prod","Sum"], [Atype, Aterm-->Atype] ---> Atype )
```

`PROD` and `SUM` relate to `Prod` and `Sum` just as `LAM` relates to `Lambda`. In particular these declarations do not result in constants being defined in the abstract syntax of `TT0`

We will not give all the definitions here. but the definition of the judgement that two terms are equal in a type (written `<| x ~= x' in X |>` or equivalently `EqElem(x,x',X)`) is:

```
("eqelem_def",
   "<| x ~= x' in X |> ==  EX u u' .(x ~> u) & (x' ~> u') & X(u,u')"),
```

The definition of the type $X + Y$ is:

```
("plus_def", "(X + Y)(x,x') == EX a a'.\
\ ((x = Inl(a)) &(x' = Inl(a')) & (<| a ~= a' in X |>)) | \
\ ((x = Inr(a)) &(x' = Inr(a')) & (<| a ~= a' in Y |>))"),
```

7.2 TT1

`TT1` is an Isabelle theory corresponding to the type theory TT_1. This has a natural interpretation in LTC_1. To build `TT1`, we need to use Isabelle's mechanism for *merging* theories. Two theories may be merged if any constants they have in common have the same type. In the merged theory, these constants are considered equal.

We first merge `LTC1` and `TT0`. Here of course, the constants that the two theories have in common are all those constants that were defined in `FOL` or `LTC0`. We then extend this theory by adding new constants and meta level equalities relating to the Universe U_1, and term forming versions of the constants of `TT0`, to get the theory `TT1`.

```
  Mixfix("<| _ ~= _ Din _ |>",[Aterm,Aterm,Aterm]--->Aterm,"DEqElem", ...
  Mixfix("DPROD _:_. _",    [SId,Aterm,Aterm]--->Aterm, " DPROD", ...
  Mixfix("DSUM _:_. _", [SId,Aterm,Aterm]--->Aterm, " DSUM", ...
  Infixr(".+", [Aterm,Aterm]--->Aterm, 30)
. . .
    (["DNat","Dvoid"],    Aterm ),
    (["DI"],             [Aterm,Aterm,Aterm] ---> Aterm ),
    (["DProd","DSum"], [Aterm, Aterm-->Aterm] ---> Aterm ),

    (["Univ"],          Alevel --> Atype ),
    (["Ty"],            Aterm --> Atype )
```

A Typical definition is:

```
("dsum_def", "DSum(a,b) == LAM z z'.DEX u u' v v'.\
\ (z .= <u,v>) .& (z' .= <u',v'>) .& \
\               (<| u ~= u' Din A |>) .& (<| v ~= v' Din b(u) |>)")
```

The axioms of TT_1 should all be derivable in the LTC once these definitions have been unfolded, we have not currently attempted this within Isabelle.

7.3 TTW

The definition of TTW follows the same lines as that of TT1. First we merge LTCW and TT1, and then extend this theory with a constant DUniv of type Alevel-->Aterm, and a meta level equality expressing the translation of $\dot{U}_i$ into the LTC.

References

[1] Peter Aczel, *Frege structures and the notions of proposition, truth and set.* In Keisler, Barwise and Kunen, editors, *The Kleene Symposium*, 31-59, North Holland, 1980.

[2] P. Aczel and P. F. Mendler, *The notion of a Framework and a framework for LTC.* In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science,* pages 392–399. IEEE, 1988.

[3] Peter Dybjer and Herbert Sander, *A Functional Programming Approach to the Specification and Verification of Concurrent Systems.* Formal Aspects of Computing (1989) 1:303–319

[4] L. C. Paulson, *The Foundation of a Generic Theorem Prover.* Technical report 130, University of Cambridge, 1988.

[5] L. C. Paulson and T. Nipkow, *Isabelle tutorial and user's manual.* Technical report 189, University of Cambridge, 1990.

[6] N. Mendler, *A series of type theories and their interpretations in the logical theory of constructions.* In these proceedings.

[7] P. Martin-Löf. *Intuitionistic type theory.* Notes by G. Sambin of a series of lectures given in Padua, June 1980, Bibliopolis, Napoli, 1984.

[8] B. Nordstrom, K. Peterson, and J. Smith. *An Introduction to Martin-Löf's Theory of Types.* OUP 1990.

[9] Jan Smith, *An interpretation of Martin-Lof's type theory in a type-free theory of propositions.* Journal of Symbolic Logic, 49:730–753, 1984.

A short description of Another Logical Framework

Lennart Augustsson, Thierry Coquand, Bengt Nordström

Department of Computer Science
University of Göteborg/Chalmers
S-412 96 Göteborg
Sweden

Abstract

This note is an incomplete description of an incomplete implementation of another logical framework.

Major characteristics

ALF is a system for editing proofs and theories. It is based on a combination of a general type system (GTS) and Martin-Löf's logical framework. It is possible to add equations when defining a theory. This possibility is implemented in the present version (it is thus possible to define the operations of Martin-Löf's monomorphic set theory inside ALF).

The most important characteristic is that ALF takes seriously the idea of "proofs as objects", in the sense that the interaction consists in building a proof-term, which is actually displayed on the screen. The major proof editing operations is to refine a placeholder (standing for an incomplete subproof) to a partial proof and to delete a partial proof, i.e. replace it with a placeholder.

The (partial) proof-term is presented together with constraints that are typing constraints (context and types of the subgoals), and equational constraints (context and pair of terms of the same type in this context). The invariant is that if all the constraints are satisfied then the (partial) term is a correct proof of the goal.

There is no direct unification algorithm. Instead, the system uses only the first part of the higher-order unification algorithm, that is the simplification of the equational constraints. These equational constraints are then displayed to the user in a simplified form. The user then control the remaining (and non deterministic) part of the higher-order unification algorithm, that is the choice of projections or imitation. So far, only a very rudimentary approximation of this is allowed, but it is enough to deal with most easy induction proofs.

The system has a tiny beginning of "structure" for theories, a theory being recursively a list of declaration of constants (primitive or defined), and theories.

The logical framework

So far ALF represents the following GTS. The sorts are Prop and Type_k, $k = 0, 1, \ldots$. The typing axioms are $\mathsf{Prop} : \mathsf{Type}_k$, and $\mathsf{Type}_i : \mathsf{Type}_j$ for $i < j$, and the typing conditions are $(s, \mathsf{Prop}, \mathsf{Prop})$

[1]This research has been done within the ESPRIT Basic Research Action "Logical Frameworks".It has been paid by STU and Chalmers.

and $(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_k)$ for $i, j \leq k$. We recall that to have the condition (s_1, s_2, s_3) means that we can form the following product

$$\frac{A : s_1 \quad B[x] : s_2\ [x : A]}{\Pi(A, B) : s_3}$$

It is intended that (following R. Pollack's "operational" presentation of GTS) , ALF will in a near future allow the user to declare any kind of GTS.

A GTS that we want to have is the one corresponding to Martin-Löf's logical framework. In this case, the sorts are Set and Type, the typing axiom is $\mathsf{Set} : \mathsf{Type}$ and the typing conditions are $(\mathsf{Type}, \mathsf{Type}, \mathsf{Type})$ and $(\mathsf{Set}, \mathsf{Set}, \mathsf{Type})$.

It is also intended to compare the "GTS approach" with a direct representation of Martin-Löf's logical framework. One important difference, for instance, is the fact that, in the latter approach, every constant has a fixed arity (and this is not the case in general in a GTS).

The theory editor

A theory is presented by a list of typings and definitions of constants. We distinguish between primitive and defined constants. A *primitive constant* has only a type, it doesn't have a definition. It gets its meaning in other ways (outside the theory). Such a constant is also called a constructor, since it computes to itself. Examples of primitive constants are N, succ and $\mathsf{0}$, where $\mathsf{N} \in \mathsf{Set}$, $\mathsf{succ} \in \mathsf{N} \rightarrow \mathsf{N}$ and $\mathsf{0} \in \mathsf{N}$.

A defined constant has a type and a definition:

$$c \in A \equiv a$$

The definiendum c computes in one step to its definiens a. A defined constant can either be explicitly or implicitly defined. An *explicitly defined constant* (macro) is just an abbreviation of its definiens (which has to be a welltyped expression). An *implicitly defined constant* is in general defined in terms of itself. Whether this is meaningful or not can only be checked outside the theory. The recursion operator over natural numbers is an example of an implicitly defined constant. When we read the constant as a name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and finally, an explicitly defined constant is a defined rule.

The system used is monomorphic. However, a constant can be declared with hidden parameters. This is only a comment for the pretty printing. In the reverse direction, we use place holders to input the hidden parameters. In this way, the user gets a polymorphic view of the system.

Groups

A problem with presenting the theory as one list of identifiens is that the list grows easily to an unmanageable size. Some structure is obtained by using the *let*-construct which makes it possible to introduce local definitions (or local lemmas). But this is not enough. We seem to need some kind of Σ-type in the framework to get more structure in the list of definitions. In the current implementation, we can put together some identifiens in a *group*, which is mainly a presentational device (it is possible to hide or show the identifiers in a group by clicking on the name of the group). A theory can be saved in a file and a group may also be stored in separate files. In our description of Martin-Löf's constructive set theory we have a group for each set-forming operation with its formation, introduction- and elimination-rules.

Commands of the theory editor

There are commands to extend a theory with new constants and groups. It is possible to include another theory (as a group) stored in a file. It is possible to move a constant between groups. It is also possible to invoke a syntax editor which makes it possible to change the concrete syntax of expressions. In the current implementation it is possible to declare a constant to be an infix, prefix or postfix operator or to be written as a quantifier.

The most important command, however is the one which invokes the proof editor which is used to edit proofs.

The proof editor

The proof editor can be described as a machine with the following registers:

- a theory (i.e. a list of definitions and typings),
- a (partial) proof term with a cursor pointing to the selection (a subproof),
- the type of the selection,
- the goal to be proven.

The following can be computed from the state of the machine:

- the type of each subterm,
- the local context, i.e. the typing of all variable bound at the selection,
- the constraints which is to hold for the place holders in order for the machine to represent a correct proof.

The screen will display all these.

The basic editing command is the one which replaces a place holder with a proof term. An example of this command is the one which applies a rule. The user then double click on one of the constants in the theory or on a variable in the local context. The system will then replace the selection (which has to be a place holder) with the term

$$c(?, ?, ..., ?)$$

where c is the name of the rule and $?, ?, ..., ?$ are enough new place holders to match the arity of the current subgoal.

The inverse command (delete) takes a selected subterm and replaces it with a new placeholder. In this way it is possible to undo arbitrary parts of the proof, and not only the most recent proof step.

There are also commands which massages the type of the current subgoal in different ways, for instance replaces a subterm of the subgoal with its definiens or its definiendum.

Proof Experiments

The formalization of Martin-Löf's monomorphic set theory is straightforward, it took only a few hours to do this.

The major experiment was done by Nora Szasz, who proved formally that Ackermann's function is not primitive recursive. The proof consists of about 200 theorems, of which around 50 are part of the proof of the growth of Ackermann's function. The rest are lemmas of properties of primitive recursive functions and natural numbers.

Acknowledgements

We want to thank the programming logic group at the department for many discussions on this topic. In particular we want to thank Lena Magnusson for her insightful comments.

The work has been done within the ESPRIT basic research action on design and implementation of logical frameworks. It has been paid by STU.

The Boyer-Moore Prover and Nuprl: An Experimental Comparison[1]

David Basin
Department of Artificial Intelligence,
University of Edinburgh, Edinburgh Scotland.
Email: basin@aipna.ed.ac.uk

Matt Kaufmann
Computational Logic, Inc.
Austin, Texas 78703 USA
Email: kaufmann@cli.com [2]

Abstract

We use an example to compare the Boyer-Moore Theorem Prover and the Nuprl Proof Development System. The respective machine verifications of a version of Ramsey's theorem illustrate similarities and differences between the two systems. The proofs are compared using both quantitative and non-quantitative measures, and we examine difficulties in making such comparisons.

1 Introduction

Over the last 25 years, a large number of logics and systems have been devised for machine verified mathematical development. These systems vary significantly in many important ways, including: underlying philosophy, object-level logic, support for meta-level reasoning, support for automated proof construction, and user interface. A summary of some of these systems, along with a number of interesting comments about issues (such as differences in logics, proof power, theory construction, and styles of user interaction), may be found in Lindsay's article [14]. The Kemmerer study [13] compares the use of four software verification systems (all based on classical logic) on particular programs.

In this report we compare two interactive systems for proof development and checking: The Boyer-Moore Theorem Prover and the Nuprl Proof Development System. We have based our comparison on similar proofs of a specific theorem: the finite exponent two version of Ramsey's theorem (explained in Section 2 below). The Boyer-Moore prover is a powerful (by current standards) heuristic theorem prover for a quantifier-free variant of first order Peano arithmetic with additional data types. Nuprl is a tactic-oriented theorem prover based on a sequent calculus formulation of a constructive type theory similar to Martin-Löf's [15]. We do not assume any prior knowledge of either system by the reader.

Why undertake such a comparison? We believe there is too little communication between those who use different mechanical proof-checking systems. This lack of communication encourages myths and preconceptions about the usability of various systems. Concrete comparisons bring to light

[1]This paper also appears as Technical Report 58 of Computational Logic, Inc. The present version is submitted for publication in the proceedings of the BRA *Logical Frameworks* Workshop '90.

[2]Earlier related work was supported by ONR Contract N00014-81-K-0634.

the limitations and advantages of different systems, as well as their commonalities and differences. Moreover, comparisons help determine which directions the field is heading and what progress is being made.[3]

Much of our comparison is based on our two proofs of the aforementioned theorem. We make qualitative comparisons, and we also make quantitative comparisons based on metrics that indicate the degree of effort required to prove the theorem. However, we caution that there is a real danger in quantitative comparisons as metrics can oversimply and mislead. Numbers cannot account for much of what is important about proof development systems and their application; e.g.,

- Expressive power of the underlying logic (e.g., first order vs. higher order, definitional extensibility, set theoretic or arithmetic, typed vs. untyped.)
- Domains particularly well-suited to verification with the given system
- Computational content (explicit or implicit), i.e., existence of executable programs associated with existence proofs (written explicitly or derived implicitly from the proof)
- Use of novel techniques in representation or in automated reasoning
- Naturality/comprehensibility of definitions and proofs, both to experienced users and to a more general community
- Ease of creating, reviewing, and changing definitions and proofs
- Proof discovery capabilities
- Soundness.

Moreover, the choice of metrics is difficult. Different metrics can favor different kinds of systems more than one might expect. For example, the number of tokens typed may not correlate with number of keystrokes if one system uses extensive cutting and pasting with an editor, or uses keyboard macros or structure editor facilities. For another example, it may or may not be the case that prover power correlates with replay time; it's not hard to envision scenarios in which a weak system's proofs replay more quickly because there is a large proof script but little heuristic search. Nonetheless, we make some quantitative comparisons:

- Lemma and definition counts
- Symbol counts
- User time required
- Replay time.

Our paper is organized as follows. Section 2 provides an informal statement and proof of the version of Ramsey's theorem alluded to above. Section 3 contains a description of the proof completed with the Boyer-Moore Theorem Prover. Section 4 contains the Nuprl proof. The final section draws comparisons and conclusions.

[3]though in fact we first did versions of these proofs at least a couple of years ago....

Acknowledgements. We thank Gian-Luigi Bellin, Jussi Ketonen, and David McAllester for a number of interesting and useful discussions on this topic. We thank Andrew Ireland, Bill Pierce, and Matt Wilding for their very helpful comments on a draft of this paper. Randy Pollack gave us some useful encouragement on this effort. David Basin also gratefully acknowledges assistance provided by Doug Howe during the Nuprl proof effort.

2 Informal Statement and Proof of Ramsey's Theorem

Ramsey's theorem is about the existence of "order" in symmetric graphs. A *symmetric graph* is a set V of vertices together with a symmetric relation E on that set. (Some formulations also require that E is irreflexive; that bears little on the essential mathematics but can bear on the details.) Let G be a symmetric graph with vertex set V and relation E. A *clique* C of G is a subset of V such that all pairs $<x, y>$ of distinct members of C belong to E; an *independent set* I of G is a subset of V such that no pairs $<x, y>$ of members of I belong to E. A set is *homogeneous* if it is a clique or an independent set. For any positive integer l we say that a subset S of V is an *l-clique* (respectively, an *l-independent set*) if it is a clique (respectively, independent set) of cardinality l. Finally, for any positive integers n, l_1 and l_2 we write

$$n \rightarrow (l_1, l_2)$$

to assert that for every graph G with at least n vertices, there is either an l_1-clique or an l_2-independent set in G.

Note: Henceforth all our graphs will be symmetric, i.e., *graph* means *symmetric graph.*

We may now state the main theorem.

Theorem. For all l_1, l_2, there exists an n such that $n \rightarrow (l_1, l_2)$.

Note: The least such n is sometimes called the *Ramsey number* corresponding to l_1 and l_2.

An informal "text-book proof" (similar to one in [9]) proceeds by double induction on l_1 and l_2. (Alternatively, the proof may proceed by a single induction on their sum — this is the tact taken in the Boyer-Moore proof.) To prove the base case observe that $l_1 \rightarrow (l_1, 2)$ as any graph with l_1 vertices either is a clique, or there are at least two vertices that are not connected by an edge. Similarly $l_2 \rightarrow (2, l_2)$. Now assume as an inductive hypothesis that we have some n and m where $n \rightarrow (l_1, l_2 - 1)$ and $m \rightarrow (l_1 - 1, l_2)$.

Claim. $n + m \rightarrow (l_1, l_2)$.

Proof. Given an arbitrary graph G on at least $n + m$ vertices, choose an element v_0 of its vertex set V. Now partition the remaining elements into two sets r_1 and r_2 where

$$\begin{aligned} r_1 &= \{x \in (V - \{v_0\}) : E(v_0, x)\} \\ r_2 &= \{x \in (V - \{v_0\}) : \neg E(v_0, x)\} \end{aligned}$$

Then $|r_1| + |r_2| = m + n - 1$ so either $|r_1| \geq m$ or $|r_2| \geq n$. If $|r_1| \geq m$, then by the induction hypothesis there is a subset S of r_1 where either S is an l_2-independent set (so we are done) or S is an $(l_1$-1)-clique. In the latter case set $S' = S + v_0$. Now, since $S \subset r_1$, there is an edge between v_0 and all $x \in S$. Hence S' is the desired l_1-clique. The case $|r_2| \geq n$ is analogous.

3 The Boyer-Moore Theorem Prover

We present our discussion of the Boyer-Moore proof in five subsections. We begin by giving enough of an introduction to the Boyer-Moore logic to be able to transform a straightforward first order formalization of the theorem into a formalization in the Boyer-Moore logic (which we also do in the first subsection). That introduction is followed by a discussion of the proof strategy. The third subsection gives a summary of the actual proof steps. We continue with some statistics and general remarks about the proof. The section concludes with a discussion of computing in the Boyer-Moore logic.

3.1 Background and Formalization of the Theorem

The Boyer-Moore theorem prover is a heuristic prover based on a simple version of a traditional first order logic of total functions, with instantiation and induction rules of inference. The logic is quantifier-free, except for the implicit universal quantification surrounding each definition and theorem. A detailed description of the logic and manual for the prover may be found in [4]. The lack of quantifiers together with the presence of induction encourages a style of specification and proof that is "constructive" in nature.[4] That is, rather than specifying the existence of various objects, one explicitly defines functions that yield those objects. The system is in fact equipped with a mechanism for computing with its defined functions (see Subsection 3.5).

The syntax of the logic is in the Lisp tradition, where the list `(f` x_1 `...` x_n`)` denotes the application of the function `f` to the arguments x_1 through x_n. We also allow the `let` form from Lisp[5], so for example the expression `(let ((x` *a*`) (y` *b*`))` *exp*`)` is equivalent to the result of substituting *a* for `x` and *b* for `y` in *exp*. Certain functions are built into the logic, such as the ordered pair constructor `cons` and the atom `nil`. The list $(x_1\ x_2\ ...\ x_n)$ is represented by the term `(cons` x_1 `(cons` x_2 `...  (cons` x_n `nil) ... ))`. The functions `car` and `cdr` select (respectively) the first and second component of a pair. Thus `car` selects the first member of a list. It is an axiom of the logic that every pair x equals `(cons (car` x`) (cdr` x`))`. Note also that since this is a logic of total functions, it makes sense to form the term `(car` x`)` for any term x, whether it is a pair (or list) or not. In practice, this lack of typing is generally not much of a problem for users once they become accustomed to it.

Consider now how one might formalize Ramsey's Theorem in this logic. If one had quantifiers then one might simply write the following. Here `pairs` is a list of pairs that represents a graph on a set `domain` of nodes, and `S` is the desired homogeneous set (i.e., clique or independent set) with

[4]Bob Boyer has pointed out that since definitions in the Boyer-Moore logic (in "thm mode", i.e., without the `V&C$` interpreter and without induction all the way up to ϵ_0) always produce primitive recursive functions, the law of the excluded middle is constructively valid for this logic when all function symbols are introduced with definitions (DEFN events).

[5]This `let` construct, implemented by J Moore, is available in the system described in [7].

respect to the graph pairs.[6]

```
     For all p and q there exists N
     such that for all domain and pairs there exists S
     such that:
     N ≤ cardinality(domain)  →
     [S ⊆ domain ∧
(a)   ((p ≤ cardinality(S) ∧ clique (pairs, S)) ∨
(b)    (q ≤ cardinality(S) ∧ independent (pairs, S)))]
```

In order to represent this conjecture in the Boyer-Moore logic, we must first eliminate the quantifiers. To that end, we will define `N` as a function `ramsey` of `p` and `q`, and we will also define `S` in terms of a function `wit` ("witness") of `p`, `q`, `domain`, and `pairs`. Actually, it is convenient to define `wit` to return an ordered pair of the form $<S, f>$, where S is the desired clique or independent set according to whether the "flag" f is 1 or 2, respectively. We'll say that `good-hom-set (pairs, domain, p, q, flg)` holds iff `flg` is 1 and disjunct (a) above holds, or else `flg` is not 1 and disjunct (b) above holds, where `S` is the first component of `wit (pairs, domain, p, q)`. The conjecture above thus transforms into the following statement.

```
     For all p, q, domain, pairs, if
         S = car (wit (pairs, domain, p, q)) and
         flg = cdr (wit (pairs, domain, p, q)), then:
             ramsey (p,q) ≤ cardinality(domain)
             →
             [S ⊆ domain ∧ good-hom-set (pairs, domain, p, q, flg)]
```

The Boyer-Moore logic has a built-in theory of lists, but not of sets. Therefore it is convenient to recast this formalization in terms of lists. We can define a predicate `setp` for a list having no duplicates. Slipping into Lisp-style syntax, we finally obtain a formalization of Ramsey's theorem. (Only the last conjunct below is new, and it says that if `domain` represents a set then so does the witness set. This is important so that the `length` of a list equals the cardinality of the set it represents; note that `length` is used in the definition of `good-hom-set`.)

```
THEOREM.
     (implies (leq (ramsey p q) (length domain))
              (and (subsetp (car (wit pairs domain p q))
                            domain)
                   (good-hom-set pairs domain p q
                     (cdr (wit pairs domain p q)))
                   (implies (setp domain)
                            (setp (car (wit pairs domain p q))))))
```

3.2 Proof Strategy

In the Boyer-Moore prover, a "proof" is a sequence of steps called *events*, typically of two kinds: definition (DEFN) events and theorem (PROVE-LEMMA) events. A major step is to define the

[6]Here p and q correspond to what were called l_1 and l_2 in the preceding section; these differ simply because each author used different variable names in their independent efforts.

function `wit`, referred to above, that constructs the clique or independent set. Unlike Nuprl, the system does not construct such a function from the proof; rather, the function is introduced by the user and its pattern of recursion is available for generating heuristic induction schemes. In fact, the proof is eventually (after some lemmas are proved) accomplished using an induction heuristically chosen by the Boyer-Moore system (see [4] or [1] for more on this topic) to reflect the recursion in the definition of the function `wit` below, i.e., by an induction on `(plus p q)`. This function returns a pair for the form `(cons set flag)` where `flag` is 1 or 2 according to whether `set` is a clique or an independent set. Everything to the right of any semicolon is a comment.

```
DEFINITION.
(wit pairs domain p q) =
(if (listp domain)
    (if (zerop p) (cons nil 1)
      (if (zerop q) (cons nil 2)
        (let ((set1 (car (partition (car domain) (cdr domain) pairs)))
              (set2 (cdr (partition (car domain) (cdr domain) pairs))))
          (if (lessp (length set1) (ramsey (sub1 p) q))
            ;; then use set2 to form clique or independent set
            (let ((wit-set2 (wit pairs set2 p (sub1 q))))
              (if (equal (cdr wit-set2) 1) wit-set2
                (cons (cons (car domain) (car wit-set2))
                      2)))
            ;; otherwise use set1 to form clique or independent set
            (let ((wit-set1 (wit pairs set1 (sub1 p) q)))
              (if (equal (cdr wit-set1) 2) wit-set1
                (cons (cons (car domain) (car wit-set1))
                      1)))))))
  (cons nil 1))
```

This definition is actually presented with a *hint* `(lessp (plus p q))` that instructs the prover to verify that the sum of the final two arguments of `wit` decreases in each recursive call of `wit`, thus guaranteeing termination. This informal description of what the prover does with that hint reflects a formal definitional principle in the Boyer-Moore logic, but we omit further discussion of this point.

In this example we use two main techniques to "discover" the proofs. One approach (the traditional one for Boyer-Moore prover users) is to start by presenting a lemma to the Boyer-Moore theorem prover. If the proof fails or if the output (a mixture of English and formulas) suggests that the proof probably won't complete successfully, then inspection of the output often suggests (to an experienced eye) useful rewrite (simplification) rules that one might wish to prove. The other main technique is to use an interactive enhancement [7] to the Boyer-Moore system as an aid to discovering the structure of the proof. This "PC-NQTHM" enhancement[7] allows one to create PROVE-LEMMA events by first submitting the proposed theorem and then interactively giving various proof commands in a backward goal-directed, "refinement" style. These commands range from "low-level" commands which invoke a particular definition or rewrite rule, to "medium-level" commands invoking simplification or (heuristic) induction, to "high-level" commands which call the Boyer-Moore theorem prover. There is also a facility for user-defined *macro commands*

[7] "PC" for "proof-checker", "NQTHM" for a common name of the Boyer-Moore theorem prover

in the tradition of the "tactics" and "tacticals" of LCF [12] and Nuprl [1]. This "proof-checker" enhancement is helpful, but not crucial, for completion of the proof. The plan was in fact to develop nice rewrite rules rather than to rely on the manual commands provided by the interactive enhancement, and this plan succeeded: the final proof contained only 10 definitions and 26 lemmas (including the final theorem) after loading the standard "ground-zero" base theory, and did not contain any proof-checker commands. The events run successfully in the unenhanced Boyer-Moore system. Other than a few hints – 7 of the lemmas took standard Boyer-Moore "hints" that specify use of one or two previously proved lemmas – the proofs are fully automatic.

3.3 Outline of Main Proof Steps

We divide this subsection into four parts: one for the requisite definitions and then one for each of the three conjuncts of the conclusion of the main theorem (see Subsection 3.1).

3.3.1 Definitions

Several definitions are necessary for the proof. One of these is the definition of the function wit, provided in the preceding subsection, which picks out the desired clique or independent set. Another important definition is of a function ramsey that provides an upper bound on the Ramsey number. Here is that definition, expressed in the official syntax with formal parameters p and q. (As mentioned above, semicolons denote comments.)

```
(defn ramsey (p q)
  (if (zerop p)
      1
      (if (zerop q)
          1
          (plus (ramsey (sub1 p) q)
                (ramsey p (sub1 q)))))
   ;; hint to the prover for its proof of termination,
   ;; suggesting that the sum of the two arguments decreases on each recursive call
  ((lessp (plus p q))))
```

The definition of wit depends not only on the function ramsey, but also depends on three other functions, which we describe here informally; see [11] for details. For any list pairs, (related i j pairs) is *true* (t) if and only if the pair <i,j> or the pair <j,i> belongs to pairs; in this way we represent the notion of symmetric binary relation. (partition n rest pairs) returns a pair <x,y> where x consists of all elements of the list rest that are related (in the sense above) to n, and y consists of the remaining elements of rest. Finally, (length lst) is the length of the list lst.

The function setp recognizes whether or not a list represents a set by returning *true* (t) if and only if the list contains no duplicates. Notice the use of primitive recursion to express bounded quantification. This is a common technique for defining universally-quantified concepts in the Boyer-Moore logic.

```
(defn setp (x)
  (if (listp x)
      (and (not (member (car x) (cdr x)))
           (setp (cdr x)))
    t))
```

The function `homogeneous` defined below recognizes whether a given set `domain` is homogeneous (i.e., a clique or independent set) for the relation (graph) represented by the list `pairs`: it tests whether `domain` is a clique if the formal parameter `flg` is 1, and otherwise it tests whether `domain` is an independent set. Notice that `homogeneous` is defined by recursion on its first argument, using an auxiliary function `homogeneous1` that checks that its first parameter `n` is related or not related (according to whether or not the parameter `flg` is 1) to every element of `domain`.

```
(defn homogeneous1 (n domain pairs flg)
  (if (listp domain)
      (and (if (equal flg 1)
               (related n (car domain) pairs)
               (not (related n (car domain) pairs)))
           (homogeneous1 n (cdr domain) pairs flg))
      t))


(defn homogeneous (domain pairs flg)
  (if (listp domain)
      (and (homogeneous1 (car domain) (cdr domain) pairs flg)
           (homogeneous (cdr domain) pairs flg))
      t))
```

Our formalization of Ramsey's theorem also requires the notion of a "sufficiently large" homogeneous set, i.e., one that has at least `p` or `q` elements depending on whether the set is a clique or an independent set.

```
(defn good-hom-set (pairs domain p q flg)
  (and (homogeneous (car (wit pairs domain p q))
                    pairs
                    flg)
       (not (lessp (length (car (wit pairs domain p q)))
                   (if (equal flg 1) p q)))))
```

Finally, we need the notion of subset. We omit here its straightforward recursive definition as well as several standard related lemmas such as transitivity.

3.3.2 The constructed set is contained in the given set

We wish to prove the following lemma. Notice the syntax for lemmas: `(prove-lemma` *lemma-name lemma-types statement*`)`, where *lemma-types* is often `(rewrite)` to indicate that the lemma is to be used in subsequent proofs as a rewrite rule. (See [4] for details.)

```
(prove-lemma subsetp-hom-set-domain (rewrite)
  (subsetp (car (wit pairs domain p q))
           domain))
```

The theorem prover proceeds by an induction suggested by the recursion in the definition of the function `wit`. The proof actually fails at first, but the prover's output suggests some lemmas. Here is one of those, which says that the second component returned by `partition` is a subset of the given set.

```
(prove-lemma subsetp-cdr-partition (rewrite)
  (subsetp (cdr (partition x z pairs))
           z))
```

A similar lemma is required for the first component, and these are used to prove two rather technical lemmas suggested by the attempted proof of the lemma SUBSETP-HOM-SET-DOMAIN. After these four lemmas, the inductive proof of SUBSETP-HOM-SET-DOMAIN succeeds without further interaction.

3.3.3 The constructed set is homogeneous and large enough

The following lemma is at the heart of the theorem.

```
(prove-lemma wit-yields-good-hom-set (rewrite)
  (implies (not (lessp (length domain) (ramsey p q)))
           (good-hom-set pairs domain p q
                         (cdr (wit pairs domain p q)))))
```

Unfortunately, the proof attempt does not succeed at first, so we must prove some supporting lemmas. This time we use the proof-checker enhancement (see Subsection 3.2) of the Boyer-Moore prover to explore the situation. Specifically, an INDUCT command is used to invoke a heuristic choice of induction scheme, a PROVE command is used in order to call the Boyer-Moore prover to dispose of the three "base cases", the function `good-hom-set` is expanded in one of the remaining four (inductive) cases, a casesplit is performed to create 8 subgoals, and finally inspection of one of those subgoals suggests the following lemma.

```
(prove-lemma homogeneous1-subset (rewrite)
  (implies (and (subsetp x domain)
                (homogeneous1 elt domain pairs flg))
           (homogeneous1 elt x pairs flg)))
```

The Boyer-Moore prover proves this lemma automatically. Manual application of this rule (in the proof-checker), to the subgoal referred to above, suggests another lemma, which is also proved automatically.

```
(prove-lemma homogeneous1-cdr-partition (rewrite)
  (homogeneous1 elt (cdr (partition elt dom pairs)) pairs 2))
```

Further attempts to prove subgoals inside the proof-checker, as well as attempts to prove the main goal WIT-YIELDS-GOOD-HOM-SET in the context of the two lemmas displayed above (and others discovered subsequently), lead to some additional lemmas. One is completely analogous to HOMOGENEOUS1-CDR-PARTITION (displayed just above), but for the `car` of the partition in place of the `cdr`. Another (also proved automatically) asserts that the cardinality of a set equals the sum of the cardinalities of the two sets returned by `partition`. Still another asserts that `ramsey` always returns a positive integer. Four other technical lemmas seem necessary for the prover's rewriter to behave properly; they are omitted here. Finally, the proof of our goal WIT-YIELDS-GOOD-HOM-SET (see above) succeeds.

3.3.4 The constructed set is indeed a set

The final goal is to prove that the homogeneous set is really a set.

```
(prove-lemma setp-hom-set (rewrite)
  (implies (setp domain)
           (setp (car (wit pairs domain p q)))))
```

The Boyer-Moore prover does not succeed in proving this automatically, but the output suggests the following useful (though obvious) lemma whose proof does succeed automatically. It states that the lists returned by `partition` are sets if the given list is a set.

```
(prove-lemma setp-partition (rewrite)
  (implies (setp x)
           (and (setp (car (partition a x pairs)))
                (setp (cdr (partition a x pairs))))))
```

Two technical lemmas discovered using the proof-checker then suffice for concluding the proof of SETP-HOM-SET.

3.4 More Statistics and General Remarks

The first time we formalized and proved this theorem was in January, 1987. The resulting proof was very ugly, as the use of the (then new) proof-checker enhancement was quite undisciplined. It seems best to find a good combination of elegant rewrite rules even when one has the capabilities offered by the proof-checker. Perhaps surprisingly, the "manual" proof takes longer to replay than the "heuristic" one reported here: total time was 219.7 seconds as opposed to 394.7 seconds for the older proof, both on a Sun 3/60 with 16 megabytes of main memory.

It took about 7 hours of human time to complete the current proof effort, which resulted in a list of events that is accepted by the Boyer-Moore prover (without the interactive enhancement). That number may be somewhat misleading since the new proof effort took advantage of the definitions and a few of the lemmas created in the earlier version. However, the more disciplined approach used in the final effort suggested changes that were made in the definitions, so it seems reasonable to guess that an effort from scratch would have taken not much longer than 7 hours anyhow. A more detailed annotated chronicle of this proof effort may be found in [11].

3.5 Computing

The Boyer-Moore system provides a mechanism for computing values of variable-free terms in the logic. This mechanism of providing *executable counterparts* to defined functions is important as part of the theorem proving process. For one thing, it allows the fast replacement of variable-free terms by constants. This was an important consideration in, for example, the operating system kernel proof described in [3], where there were many large variable-free terms in the proof; compiling the executable counterparts sped up the proof process significantly.[8] Fast execution is also valuable when executing *metafunctions* [2], which are functions that are written in the logic for the purpose of simplifying terms and can be used as code once they are proved correct.

In this subsection we address the use of the Boyer-Moore logic as a general purpose programming language. For example, if we want a homogeneous set of cardinality 3, execution of the function `ramsey` tells us that such a set exists within any graph with at least 20 vertices:

```
>(r-loop)  ;; We type this in order to enter the Boyer-Moore reduction loop.
Trace Mode: Off   Abbreviated Output Mode:  On
Type ? for help.
*(ramsey 3 3)  ;; We ask the system to compute the value of  ramsey on inputs 3 and 3.....
 20
*
```

This is however an unsatisfactory answer, given that the Nuprl proof (see next section) provides a value of 6 rather than 20 in this case. Therefore, we have in fact re-done the proof using slightly different definitions of the functions `ramsey` and `wit` that more closely reflect the induction that takes place in the Nuprl proof, which is grounded at 2 rather than at 0. We then obtain 6 rather than 20 for the value of `(ramsey 3 3)`. It took roughly 5 hours to redo the proof for these new versions of `ramsey` and `wit`.

Convention. In the remainder of this subsection we refer to the alternate versions of `ramsey` and `wit` mentioned above.

The (new version of the) function `wit` can also be executed, on a particular set and binary relation, with particular values for its parameters `p` and `q`. Consider for example the hexagon, represented as a binary relation connecting i to $i+1$ for i from 1 to 6 (except that 6 is connected to 1).

```
*(wit '((1 . 2) (2 . 3) (3 . 4) (4 . 5) (5 . 6) (6 . 1))
      '(1 2 3 4 5 6)
      3 3)
 (CONS '(1 3 5) F)
*
```

Thus, the set $\{1,3,5\}$ is an independent set, as indicated by the second component `F` of the value returned above (which corresponds to `2` in the version discussed in the rest of this paper). We found on a Sun 3/60 that it took an average of about 0.15 seconds to evaluate this `wit` form. However, after compiling an appropriate file, the time was reduced to 0.011 seconds per form.

[8] personal communication from Bill Bevier

4 Nuprl

Our presentation of the Nuprl proof is divided into four subsections. The first provides a brief overview of Nuprl. The reader is encouraged to consult [1] for further details. The second summarizes definitions, theorems, and tactics used in our proof. The third presents main proof steps. And the final subsection documents the computational content of our proof.

4.1 Background

The basic objects of reasoning in Nuprl are types and members of types. The rules of Nuprl deal with *sequents*, objects of the form

$$x_1{:}H_1,\ x_2{:}H_2,\ ...,\ x_n{:}H_n >> A.$$

Informally, a sequent is true if, when given members x_i of type H_i (the *hypotheses*), one can construct a member (*inhabitant*) of A (the *goal* or *conclusion*). Nuprl's inference rules are applied in a top down fashion. That is, they allow us to refine a sequent obtaining subgoal sequents such that a goal inhabitant can be computed from subgoal inhabitants. Proofs in Nuprl are trees where each node has associated with it a sequent and a refinement rule. Children correspond to subgoals that result from refinement rule application.

These refinement rules may be either primitive inference rules or ML programs called tactics. Nuprl tactics are similar to those in LCF [12]: given a sequent as input, they apply primitive inference rules and other tactics to the proof tree. The unproved leaves of the resulting tree become the subgoals resulting from the tactic's application. Tactics act as derived inference rules; their correctness is justified by the way the type structure of ML is used.

Nuprl's type theory is expressive; its intent is to facilitate the formalization of constructive mathematics. Higher order logic is represented via the propositions-as-types correspondence. Under this correspondence an intuitionistic proposition is identified with the type of its evidence or proof objects. For example, an intuitionistic proof of $A \Rightarrow B$ is a function mapping proofs of A to proofs of B, i.e., a member of the function space $A \rightarrow B$. Similarly, a proof of $A\&B$ inhabits the cartesian product type $A\#B$. A proposition is true when the corresponding type is inhabited. For example, $\lambda x.\, x$ is a member of the true proposition $A \Rightarrow A$. Types are stratified in an unbounded hierarchy of universes beginning with U_1. One may quantify over types belonging to a given universe, but the resulting type (predicatively) belongs to a higher universe.

To prove a proposition P of constructive mathematics in Nuprl, one proves "$>> T$" for the appropriate type T by applying refinement rules until no unproven subgoals exist. If P is true, a proof will produce a member of T (the proof object) that embodies the proof's computational content. Nuprl provides facilities to extract and execute this content. Thus, Nuprl may be viewed as a system for program synthesis: Theorem statements are program specifications, and the system extracts proven correct programs.

Theorem proving takes place within the context of a Nuprl *library*, an ordered collection of tactics, theorems, and definitions. Objects are created and modified using window-oriented, structure editors. Nuprl contains a definition facility for developing new notations in the form of templates (display forms) which can be invoked when entering text. Notations are defined using other definitions and ultimately terms within the type theory. Required properties (or axioms) about defined objects may not be assumed; they must be proved within the theory.

4.2 Theory Development

Our proof of Ramsey's theorem required approximately two weeks of work. Most of this time was spent building a library of foundations for reasoning about finite sets and graphs. Our self-contained library contains 24 definitions and 25 lemmas (counting the final theorem and excluding "ground zero" definitions such as the direct encodings of the logical connectives of predicate calculus). Nineteen definitions and all of the lemmas are related to finite sets, four definitions are related to graphs, and one definition is relevant to the statement of Ramsey's theorem. A complete list of definitions and lemmas may be found in [1].

Our library is built in a rather general way and has been used by researchers at Cornell (in addition to the first author) to prove theorems in graph theory. Rather than assuming that finite sets are built from some specific type (such as integers) most of our definitions and theorems are parameterized by a type whose members are types with decidable member equalities. This type of types (which we refer to as the type of "discrete type pairs" and display as D) is defined as

$$T : U_1 \# \forall x, y : T.\, x = y \textit{ in } T \vee \neg(x = y \textit{ in } T).$$

For example, the type of integers (int) paired with a decision procedure for member equality belongs to this type D. If A is a member of D, we denote the first projection (or carrier) of A by $|A|$. Most definitions (e.g., finite set membership $\epsilon\{A\}$ or finite set subset $\subset\{A\}$) and theorems carry along a reference to this type parameter in their statement. Such generality is not required for Ramsey's theorem as we could have fixed a specific discrete type, such as the integers, throughout the proof; hence, in the presentation that follows, we often leave this parameter (A) implicit. A full discussion of the benefits of such parameterization may be found in [1].

Finite sets (displayed as $FS(A)$) are defined as lists without duplication, containing members from some discrete type A. Specifically, finite sets are defined as the parameterized (over a discrete type pair A) type[9]

$$\lambda A.\, \{l : |A|\ list \mid [nil \to True; h.t, v \to \neg(h\,\epsilon\{|A|\}\,t)\ \&\ v;\ @l]\}.$$

whose members are empty lists (the base case, when l is the empty list, evaluates to $True$), and all non-empty $|A|$ lists whose heads are not members of their tails and the same is recursively required of their tails. Given this definition, finite set membership is defined in the obvious way as list membership, subset in terms of finite set membership, cardinality as list length, etc.

Many of the theorems we prove about finite sets read like set theory axioms, and their proofs implicitly construct useful functions on finite sets. Consider, for example, the following lemma, which states that the union of two finite sets is a finite set: "for all A of the type D of discrete type pairs, and for all r_1 and r_2 that are finite sets over $|A|$, there exists a finite set r over $|A|$ such that for all x in $|A|$, x belongs to r if and only if x belongs to either r_1 or r_2."

$$\forall A : D.\, \forall r_1, r_2 : FS(A).\, \exists r : FS(A).\, \forall x : |A|.\, x\,\epsilon\{A\}\,r \Leftrightarrow x\,\epsilon\{A\}\,r_1 \vee x\,\epsilon\{A\}\,r_2$$

As its proof must be constructive, it provides an actual procedure that given a discrete type pair A, and two finite sets r_1 and r_2, returns a pair[10] where the first component is the union of r_1 and r_2

[9] The term $[nil \to b; h.t, v \to w;\ @l]$ is defined as (a hopefully more readable version of) Nuprl's list recursion combinator $list\_ind(l; b; h, t, v.w)$. When l is nil this term reduces to b and when l is $h'.t'$ it reduces to the term w' where w' is w with h' substituted for (free occurrences of) h, t' substituted for t, and $list\_ind(t'; b; h, t, v.w)$ substituted for v.

[10] A constructive proof of $\exists x.P$ is a pair $<a, p>$ where p proves $P[a/x]$.

```
Graph:
A:D # V:FS(A) # E: (elts{A}(V)->elts{A}(V)->U1) #
   ∀x,y:elts{A}(V). E(x,y) ∨ ¬E(x,y) &
   ∀x,y:elts{A}(V). E(x,y) <=> E(y,x) &
   ∀x:elts{A}(V). ¬E(x,x)

n→ (l1,l2):
λ n l1 l2. ∀G:Graph. n ≤ |V(G)| =>
  ∃s:FS(|G|). s ⊂ {|G|} V(G) &
    |s| = l1 & ∀x,y ϵ{|G|} s. ¬(x ={||G||} y) => E(G)(x,y)  ∨
    |s| = l2 & ∀x,y ϵ{|G|} s. ¬E(G)(x,y)

ramsey:
>> ∀l1,l2:{2..}. ∃n:N+. n→ (l1,l2)
```

Figure 1: Graph and Ramsey Theory

and the second is constructive evidence of this fact. This procedure is supplied automatically from the completed proof by Nuprl's extractor. An additional library object is created that associates a definition for finite set union (displayed as $\cup\{A\}$) with this extraction. Another typical example is the lemma *pick* which states that an element can be picked from any non-empty finite set.

$$\forall A\!:\!D.\, \forall s\!:\!FS(A).\, 0 < |s| \Rightarrow \exists x\!:\!|A|.\, x \,\epsilon\{A\}\, s$$

Several additional definitions leading up to the statement of Ramsey's theorem are provided in Figure 1. The first defines the type of graphs (displayed as *Graph*). A graph is parameterized by a discrete type pair A, and contains a vertex set V, and an edge relation E that is decidable, symmetric, and irreflexive. Not shown are projection functions that access the graph's carrier, vertex set, and edge relation components. Their display forms are $|G|$, $V(G)$, and $E(G)$ respectively.[11] The second definition defines a "ramsey function" (displayed as $n \rightarrow (l_1, l_2)$), which states that any graph with at least n vertices contain an l_1-clique or an l_2-independent set.[12] The third object is the statement of Ramsey's theorem itself. $\{2..\}$ and $N+$ represent the set of integers at least two and the positive integers, respectively.

Our development of finite set and graph theory also includes building specialized tactics. Nuprl's standard tactic collection [17] contains a number of modules for automating common forms of logical reasoning. On top of these are special purpose tactics for set theoretic reasoning. The most powerful of these, *FSTactic*, uses a combination of term rewriting, backchaining, propositional reasoning, and congruence reasoning to solve set membership problems. For example, the lemma defining finite set union is automatically used to rewrite membership in a union of sets to a disjunction of membership terms. Our collection consists of about 150 lines of ML code and was written in a day.

[11] Note that we have overloaded display forms (e.g., we have two distinct definitions that display the same way) such as vertical bars which project carriers from both graphs and discrete type pairs. Hence, in the second definition in Figure 1, the term $||G||$ is the type given by projecting out the discrete type pair from a graph tuple G and then projecting out the type from the resulting pair.

[12] We have used indentation to aid the reader in parsing such expressions. In the actual system, a structure editor is used that has facilities for disambiguating parsing when creating and viewing expressions.

4.3 Outline of Main Proof Steps

The actual Nuprl proof of Ramsey's theorem closely follows the informal outline in Section 2, albeit with more detail. The following snapshots represent the main steps and are taken from the actual Nuprl session.[13] These steps convey much of the flavor of proof development in Nuprl and indicate how carefully developed definitions, lemmas, and tactics, facilitate high-level, comprehensible proof development.

The first snapshot is of the initial proof step. At the top is the goal, our statement of Ramsey's theorem. It is followed by a refinement rule, a combination of tactics that specifies induction on l_1 using j_1 as the induction variable. The next two lines contain the hypothesis and subgoal for the base case, and the last three lines contain the inductive case. This simple looking refinement step hides 64 primitive refinement rules.[14]

```
>> ∀l1,l2:{2..}. ∃n:N+. n→ (l1,l2)
BY OnVar 'l1' (NonNegInd 'j1')
j1 = 2
   >> ∀l2:{2..}. ∃n:N+. n→ (j1,l2)
2 < j1
∀l2:{2..}. ∃n:N+. n→ (j1-1,l2)
   >> ∀l2:{2..}. ∃n:N+. n→ (j1,l2)
```

We continue by refining the base case using the tactic *ITerm* ("Instantiate Term"), which provides the witness l_2 for n. Notice some that abbreviations are expanded after this refinement, such as $n \to (j_1, l_2)$.

```
BY ITerm 'l2'
G:Graph
l2 ≤ |V(G)|
   >> ∃s:FS(|G|). s ⊂ V(G) &
        |s| = j1 & ∀x,y ϵ s. ¬(x=y in ||G||) => E(G)(x,y) ∨
        |s| = l2 & ∀x,y ϵ s. ¬E(G)(x,y)
```

The result is that, given an arbitrary graph G with at least l_2 vertices, we must find a subset s of its vertex set such that G restricted to s is a 2-clique or an l_2-independent set. To construct such an s, we use a lemma that states that for any decidable predicate P on pairs of elements from some finite set, either P holds for all pairs, or there is a pair for which P fails. This lemma, whose formal statement is

```
>> ∀s:FS(A). ∀P:elts(s)->elts(s)->U1.∀x,y ϵ s.
      P(x)(y) ∨ ¬P(x)(y)
         => ∀x,y ϵ s. P(x)(y) ∨ ∃x,y ϵ s. ¬P(x)(y),
```

[13] Space saving simplifications have been made: Some hypotheses and uninteresting parts (e.g., applications of *FSTactic*) of refinement steps are omitted. Hypothesis numbers are omitted unless referenced in subsequent snapshots. As the proof references only one discrete type pair, such references are dropped whenever possible. For example, expressions like $s_1 \cup\{A\}\, s_2$ and $x\,\epsilon\{A\}\, s$ are replaced with the simpler $s_1 \cup s_2$ and $x \,\epsilon\, s$. The complete unaltered proof is found in [1].

[14] That is, the tactics in this refinement step internally generate a 64 node proof tree containing only primitive refinement rules. The size of this underlying proof tree, to a first approximation, roughly indicates the degree of automated proof construction.

is proved by providing (implicitly via an inductive proof) a search procedure that applies P to all x and y in a given s and returns a proof that either P holds for all x and y or returns a pair for which P fails and a proof of $\neg P(x)(y)$. Instantiating P with the appropriate edge relation justifies the following.

```
BY Cases ['∀x,y ∊ V. (λ z w. z = w in |A| ∨ ¬E(z,w))(x)(y)';
          '∃x,y ∊ V. ¬((λ z w. z = w in |A| ∨ ¬E(z,w))(x)(y))']
```

When the first case holds, we are provided with a proof of $\neg E(x, y)$, for all x and y in V such that $x \neq y$. As $|V| \geq l_2$, we can pick a subset of V that is an l_2-independent set. In the second case, we are given an x and a y that have an edge between them. Hence, this subset of V is a 2-clique under the edge relation E. It takes 11 refinement steps to complete this analysis.

To prove the inductive case of the first induction, we perform a second induction.

```
BY OnVar `l2` (NonNegInd `j2`)
j2 = 2
   >> ∃n:N+. n→ (j1,j2)
2 < j2
∃n:N+. n→ (j1,j2-1)
   >> ∃n:N+. n→ (j1,j2)
```

The proof of the base case of this second induction is analogous to the first base case, and we say no more about it here. We now have two induction hypotheses, which, after several elimination steps, are as follows.

```
(6) n→ (j1,j2-1)
(8) m→ (j1-1,j2)
```

These furnish the required Ramsey number for the second induction step.

```
>> ∃n:N+. n→ (j1,j2)
BY ITerm 'n + m'
G:Graph
n+m ≤ |V(G)|
   >> ∃s:FS(|G|). s ⊂ V(G) &
        |s| = j1 & ∀x,y ∊ s. ¬(x=y in ||G||) => E(G)(x,y) ∨
        |s| = j2 & ∀x,y ∊ s. ¬E(G)(x,y)
```

After expanding G into its constituent components (a discrete type pair A, a vertex set V, an edge relation E, and edge properties p_1, p_2, and p_3 — i.e., the six parts of the graph tuple defined in Figure 1), we instantiate the outermost quantifiers of the *pick* lemma to select an element v_0 from V. Then, using finite set comprehension, provided by a lemma *fs_comp* which states that any set (here $V - v_0$) may be partitioned by a decidable property (here $\lambda x.\, E(v_0)(x)$), we divide V into r_1 and r_2: those elements of V connected to v_0 and those not.

```
BY InstLemma `pick` ['A';'V'] THEN InstLemma `fs_comp` ['A';'V - v0';'E(v0)']
```

This leaves our conclusion unchanged and provides the following new hypotheses.

```
v0 ∊ V
(19) ∀x:elts(V - v0). x ∊ r1 <=> x ∊ V - v0 & E(v0,x) &
                      x ∊ r2 <=> x ∊ V - v0 & ¬E(v0,x)
disj(r1,r2)
r1 ∪ r2 = V - v0
```

Using the last two hypotheses and our cardinality lemmas, we prove that $|r_1| + |r_2| \geq m + n - 1$. This takes two refinement steps. It follows (using a tactic for simple monotonicity reasoning) that $|r_1| \geq m \vee |r_2| \geq n$, and we split on the two cases. In the $|r_1| \geq m$ case, we instantiate hypothesis 8, one of our two induction hypotheses as follows.

```
BY EOnThin '<A,<r1,<E,<p1,<p2,p3>>>>>' 8
```

Recall that a Graph is a tuple. The tactic *EOnThin* ("Eliminate On and then Thin (drop) the indicated (uninstantiated) hypothesis") performs an elimination step that instantiates hypothesis 8 with the graph G restricted to the vertex set $r_1 \subset V$. The other components of G are as before. This yields the following hypothesis.

```
(m ≤ |r1|) => ∃s:FS(A). s ⊂ r1 &
   |s| = j1-1 & ∀x,y ∊ s. ¬(x=y in |A|) => E(x,y) ∨
   |s| = j2 & ∀x,y ∊ s.¬E(x,y)
```

Breaking down this hypothesis yields another case split. In the first case, we are given the new hypotheses:

```
(22) s ⊂ r1
|s| = j1-1
(26) ∀x,y ∊ s.¬(x=y in |A|) => E(x,y)
```

Our conclusion remains unchanged. We must still produce a subset s of V that contains a j_1-clique or a j_2-independent set. In this case we prove the left disjunct by introducing the set $s + v_0$ and demonstrate that it constitutes a clique.

```
>> ∃s:FS(A). s ⊂ V &
     |s| = j1 & ∀x,y ∊ s. ¬(x  = y in |A|) => E(x,y) ∨
     |s| = j2 & ∀x,y ∊ s. ¬E(x,y)
BY ITerm 's + v0' THEN ILeft
s + v0 ⊂ V
|s + v0| = j1
x ∊ s + v0
y ∊ s + v0
(33) ¬(x=y in |A|)
   >> E(x,y)
```

This step results in the new goal of proving that G restricted to $s + v_0$ is indeed a clique: that is, given an arbitrary x and y in $s + v_0$, proving $E(x, y)$. The first two hypotheses in the display above in fact correspond to two other subgoals produced by the indicated refinement; however, they are proved automatically by the tactic *FSTactic* mentioned previously.[15] This new goal is proved by

[15]This does not come for free. *FSTactic* undertakes significant amounts of search and this is slow. Its execution time contributes significantly to our replay time and library development time discussed in the next section.

examining the four possible cases of $x = v_0$ and $y = v_0$. This type of reasoning is routine and each case takes one refinement step (invoking the suitable tactic or combination of tactics) to verify. If $x = y = v_0$, then this contradicts hypothesis 33. If neither are, than both vertices are in s and $E(x, y)$ follows from hypothesis 26. In the remaining two cases, one vertex is v_0 and the other is not; $E(x, y)$ follows from hypotheses 19 and 22, together with the definition of r_1 and the symmetry of E.

In the second case, our new hypotheses are

```
s ⊂ r1
|s| = j2
∀x,y ϵ s. ¬E(x,y)
```

Instantiating the goal with s now proves the existence of a j_2-independent set.

```
>> ∃s:FS(A). s ⊂ V &
     |s| = j1 & ∀x,y ϵ s. ¬(x=y in |A|) => E(x,y) ∨
     |s| = j2 & ∀x,y ϵ s. ¬E(x,y)
BY ITerm 's' THEN IRight
```

FSTactic completes the $|r_1| \geq m$ case. The other case is proved analogously.

Overall, the entire proof consists of 64 refinement steps and took about 20 hours to prove, including aborted proof attempts. Tactics played a major role in making this development feasible; the 64 refinement steps hide 17531 primitive steps, an expansion factor of 273 to 1.

4.4 Computational Content

Although our constructive proof may be more complicated than a corresponding classical proof, our proof constructs three interesting functions that can be automatically synthesized by Nuprl's extractor. They are displayed below. Let us note that *term_of(thm)* is a term that evaluates to the extraction from a theorem *thm*, i.e., evaluates to a proof of *thm*. ".1" and ".2" are defined as first and second projection functions on pairs.

ram_n The outermost type constructors of Ramsey's theorem define an AE ($\forall/\exists$) formula. Hence, its extraction, $\lambda l_1 l_2.\mathit{term\_of}(\mathit{ramsey})(l_1)(l_2).1$, constitutes a function from integers l_1 and l_2 that evaluates to the first projection of a pair $<n, P_1>$. This function returns n, an upper bound on the Ramsey number for l_1 and l_2.

ram_clique P_1, the second component of the above pair, is the computational part of our proof that $n \rightarrow (l_1, l_2)$. This too is an AE statement (see Figure 1) and defines a function whose application to a G in *Graph* and a trivial proof (*axiom*) that G is sufficiently large evaluates to $<s, P_2>$. The function $\lambda l_1 l_2 g.\mathit{term\_of}(\mathit{ramsey})(l_1)(l_2).2(g)(\mathit{axiom}).1$, returns the set s, where G restricted to s is an l_1-clique or an l_2-independent set.

ram_decide P_2 is the computational content of our proof of a disjunction. It provides the basis for a procedure that prints "Clique" or "Independent Set" depending on which disjunct holds.

Nuprl contains facilities to execute these functions. For example, let G be the graph isomorphic to a hexagon.

$$G = \mathit{make\_graph}((2.6.\mathit{nil}).(1.3.\mathit{nil}).(2.4.\mathit{nil}).(3.5.\mathit{nil}).(4.6.\mathit{nil}).(5.1.\mathit{nil}).\mathit{nil})$$

	Boyer-Moore	Nuprl
# Tokens	933	972
# Definitions	10	24
# Lemmas	26	25
Replay Time	3.7 minutes	57 minutes

Figure 2: Comparison Statistics

Here, *make_graph* is a routine that converts an adjacency list (i.e., vertex 1 has an edge to vertex 2 and an edge to vertex 6, ...) to a tuple of type *Graph*. Execution of *ram_n*(3)(3) returns six, so G contains three vertices that constitute a clique or an independent set. Execution of *ram_clique*(3)(3)(G) returns the set 6.(2.(4.nil)) and *ram_decide*(3)(3)(G) prints *Independent Set*. Their execution takes about seven seconds on a Symbolics 3670 Lisp Machine.

5 Comparison

At the heart of the differences between the Boyer-Moore theorem prover (BMTP) and Nuprl are the philosophies underlying the two systems. BMTP is poised on a delicate balance point between logical strength and theorem proving power. The logic is just strong enough to express many interesting problems, but not so strong that proof automation becomes unmanageable. In the case of Ramsey's theorem, the BMTP proof was essentially constructive by necessity: as the logic lacks quantifiers, "existence" cannot be expressed other than by providing witnessing objects in the theorem statement. Of course, the upside is that around such a restricted logic, Boyer and Moore have been remarkably successful at designing heuristics for automating theorem proving; this is perhaps reflected in the quantitative comparison in Figure 2,[16] and it is certainly reflected in some large proof efforts that have been carried out using that system and PC-NQTHM (see for example [2] and the other articles on system verification in that issue of the Journal of Automated Reasoning). It is also reflected in the total times for the proof efforts. The Nuprl effort took about 60 hours for library development and about 20 additional hours to complete the proof.[17] The Boyer-Moore effort took about 7 hours altogether, though (as explained in Subsection 3.4) a few hours may have been saved because of the existence of a previous proof. Finally, the simplicity and classical nature of the Boyer-Moore logic makes it quite accessible to those with little background in logic.

The philosophy behind Nuprl is to provide a foundation for the implementation of constructive mathematics and verified functional programs. Within this framework, the responsibility for automating reasoning falls almost entirely upon the user, though Nuprl does contain a built-in

[16]In that figure, "# Tokens" refers to (essentially) the total number of identifiers and numerals in the input submitted to the systems. It does not count Nuprl tactics or Nuprl definitions or theorem statements leading up to the final proof. Replay times refer to runs on a Sun 3/60 using akcl for the BMTP run and a Symbolics 3670 Lisp Machine for the Nuprl run. The Nuprl replay time measures the time required to expand (i.e., produce a tree of primitive refinement rules) only the proof of Ramsey's theorem. Nuprl version 3.0 was used for these measurements; a new version is soon to be released that is substantially (up to a factor of 2) more efficient.

[17]Much of this time was spenting waiting for Nuprl and would be saved by a more efficient (see previous footnote) version of the system.

decision procedure for a fragment of arithmetic and Nuprl's standard tactic collection provides significant theorem proving support. Moreover, the logical complexity of Nuprl's type theory is reflected in the complexity of the tactics. For example, term well-formedness is not merely a syntactic property of terms; it is a proof obligation that is undecidable in general, as it is equivalent to showing that a program meets its specification. Nuprl's standard tactic collection contains procedures that in practice solve well-formedness problems (i.e., that a term belongs to some universe), but nonetheless well-formedness is an additional burden on the user and tactic writer and is reflected in development and replay time. However, the richness of the logic contributes to the ease with which problems may be formulated. And its constructivity enables the system to construct interesting programs as results of the theorem-proving process.

It is difficult to compare the naturality or the ease with which one finds proofs in different systems — especially when the theorem proving paradigms are as different as BMTP and Nuprl's. In BMTP, the user incrementally provides definition and lemma statements that lead up to the desired theorem statement. Each lemma is proved automatically, as is the final theorem. Interaction with the system consists of the user analyzing failed proof attempts and determining what intermediate lemmas, and perhaps hints, are needed to help the prover find a proof. Refinement style proof is possible using the PC-NQTHM interactive enhancement. On the other hand, Nuprl proofs are constructed entirely interactively by refinement.

Interestingly, approximately the same number of lemmas were used for both proofs. However, the lemma statements tend to be rather different. Almost all of the Nuprl lemmas are general purpose propositions about finite sets and are not suggested by failed automated proof attempts. As a result, their statements are often more intuitive than the more technical of the lemmas used in the BMTP proof. Moreover, the Nuprl proofs seem syntactically close to the style in which mathematics is traditionally presented and provide a formal document of why a theorem is true. Let us return to the question of what is learned from counting the number of lemmas. The sequence of Nuprl refinement steps is probably a more natural analogue of the sequence of Boyer-Moore lemmas than is the sequence of Nuprl lemmas. For, Boyer-Moore lemmas and Nuprl refinement steps are all atomic steps carried out automatically without user interaction, and they all take advantage of reasoning capabilities provided by the system (including the tactic libraries, in the case of Nuprl).

Another interesting point, related to proof style, is that, to a certain degree, Nuprl proofs should *not* in general be completely automated. Although any proof is sufficient to demonstrate that a theorem is true, different proofs have different computational content. When algorithmic efficiency is a consideration, the user requires control over the proof steps that affect program complexity. This is not an issue with BMTP proofs as programs are explicitly given in definitions.

Both Nuprl and the Boyer-Moore prover have strong connections with programming, although the approaches are different. In Nuprl, one may directly verify that an explicitly given program (a term in Nuprl's type theory) meets a specification. However, programs are usually extracted as the implicit "computational content" of proofs. Subsection 4.4 provides examples of the latter approach. In the Boyer-Moore system one must explicitly provide the programs (though they are usually called definitions or "DEFN events") in a language closely related to pure Lisp. However, those programs are translated quite directly by the system into Common Lisp code, which can then be compiled. Therefore these programs can be considerably more efficient than the unoptimized, interpreted programs extracted by Nuprl. (Recall the numbers in Subsections 3.5 and 4.4: 0.15 seconds to evaluate a Boyer-Moore form (0.011 if the functions are compiled) corresponds to about 7 seconds for evaluation of an analogous Nuprl form, albeit on different machines.) But we can envision improvements in both systems. The Nuprl term language defines a pure functional programming language that is currently interpreted in a straightforward way. It seems reasonable to assume that (with enough effort) execution could be made as efficient as current ML or Lisp implementations. Furthermore, extracted code is currently unoptimized. Work by Sasaki [17] on an optimizer for an earlier version of Nuprl indicates that type information in proofs can be utilized to extract significantly more efficient code. Improvements are also planned for program development in the Boyer-Moore system. Boyer and Moore are developing a successor to BMTP called `acl2` that will be based on an applicative subset of Common Lisp. Common Lisp is actually used for serious applications and we expect `acl2` to be more useful for programmers than the current language.

We have spent little time addressing soundness issues. Both systems are based on well-defined formal logics, so at the very least it is possible to ask whether the systems do indeed implement their respective logics. Both systems have been crafted sufficiently carefully and used sufficiently extensively to give us some confidence that when a statement is certified by the system as being a theorem, then it is indeed a theorem. Another soundness issue is the extent to which a system helps users to develop specifications that reflect their informal intentions. The expressiveness of Nuprl's type theory and the ability to create new notions allow users to express specifications in a reasonably naturally and hierarchical way that follows their logical intuitions. Soundness is not compromised by definitions as the definition facility is essentially a macro facility. Furthermore the "strong typing" in Nuprl prevents certain kinds of specification errors that are analogous to errors prevented by strong typing in programming languages (e.g., array subscripts out of range). The Boyer-Moore logic (hence the prover as well) does allow bona fide definitions, even recursive ones, but guarantees conservativity and hence consistency of the resulting theories. The text printed out by the Boyer-Moore prover is also occasionally useful for discovering errors in specifications. For example, experienced users sometimes detect output indicating obviously false subgoals. Similarly, their suspicions may be raised by proofs that succeed "too quickly".

We have also said little in our comparison about user interface issues. However, in practice, actual usability depends greatly on something less glamorous than the logic or the automated reasoning heuristics, namely the editor. Serious users of BMTP tend to rely heavily on the capabilities offered by an Emacs editor. Nuprl provides special purpose editors for creating and manipulating definitions, tactics, and proofs.

We should re-emphasize that the numbers presented above, as with many metrics, are potentially misleading. Consider, for example, the problem of comparing sizes of related proofs. We measure this by counting user-entered tokens. But even this simple metric is problematic. Our count measures the number of tokens to be input for the completed proof, and hence ignores the issue of how many tokens were entered on misguided parts of the attempt that never found their way into the final proof. (An attempt to measure this total number of tokens might be informative but might well measure a user's style more than it would measure something about a particular system.) Moreover, even the final token count may be quite dependent on the way one goes about doing the proof, and this can vary wildly among users of a given system. For example, in the Nuprl system, any sequence of proof steps can be encoded as a tactic. In the Nuprl effort, we did not count the tactic code in Figure 2 since the tactics' development was for the most part fairly general.[18] Similarly, arbitrarily large chunks of the proof can be proved separately as lemmas that can be

[18] As with the rest of the finite set library, they have been used by other students at Cornell to prove other theorems in graph theory.

incorporated into the final proof. Nuprl's definition mechanism also allows text to be bound to a single tokens. Comparable difficulties exist in collecting statistics about the Boyer-Moore theorem prover. Many of the tokens come from technical lemmas whose statements were constructed by using the editor to cut and paste; hence the token count is not necessarily reflective of the amount of user interaction. And as with Nuprl, it is easy to shorten the proof with tricks, especially if we use the `let` construct, which is included with the PC-NQTHM interactive enhancement of the Boyer-Moore prover, to share subexpressions. Of the lemmas proved, six (with 65 tokens altogether) were purely about sets, and two were technical lemmas that were created easily using the Emacs editor (without typing many characters, to our recollection) but which accounted for 116 tokens; and all of these were included in the statistics.

In a similar vein, let us point out that a higher definition count might reflect a lack of expressiveness of the logic, or it might instead reflect an elegance of style (modularity). Similarly, the number of lemmas in an interactive system can reflect the user's desire for modularity or it can reflect the weakness of the prover. Replay time can measure the power of the prover, but instead it may measure the extent to which the system "saves" the decisions that were made during the interactive proof of the theorem.

Of course, in our proofs we tried not to exploit such possibilities. Furthermore, to a certain extent, "cheating" in one area is reflected in another. For example a proof may be shortened by increasing definitions and lemmas, and replay time may be decreased by increasing the size (explicitness) of the proof. Hence, quantitative measurements should all be taken together along with the non-quantitative proof aspects in order to aid understanding of the systems in question.

Perhaps the most important point we can make in closing this comparison is that both of us felt comfortable in using the system that we chose, and we each find the other system to be reasonably natural but difficult to imagine using ourselves. It seems to be the case that it takes most people at least a few weeks to get comfortable enough with a proof-checking environment in order to reasonably assess its strengths and weaknesses. We'd like to hope that the descriptions of the proof efforts presented in this paper, together with our comments about the systems, suggest that both systems are quite manageable once one invests some time to become familiar with them. We'd also like to hope that the various warnings presented in this paper encourage people to be cautious about making too-easy judgments regarding the relative merits of systems.

We would be interested in seeing more comparisons of proof development systems. Perhaps though it would be responsible of us to point out that it took much more effort to write this paper than to carry out the proofs.

References

[1] David A. Basin. *Building Theories in Nuprl.* Technical Report 88-932, Cornell University, 1988.

[2] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. "An Approach to Systems Verification." Journal of Automated Reasoning, November, 1989.

[3] William R. Bevier. "Kit: A Study in Operating System Verification." IEEE Transactions on Software Engineering, November, 1989, pp. 1368-81.

[4] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, New York, 1979.

[5] Robert S. Boyer and J Strother Moore. "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures." In *The Correctness Problem in Computer Science*, ed. Robert S. Boyer and J Strother Moore, Academic Press, London, 1981.

[6] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[7] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[8] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[9] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey Theory.* John Wiley and Sons, 1980.

[10] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, 1988.

[11] Matt Kaufmann. *An Example in NQTHM: Ramsey's Theorem.* Internal Note 100, Computational Logic, Inc., November 1988.

[12] Matt Kaufmann. *A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover.* Technical Report CLI-19, Computational Logic, Inc., May 1988.

[13] Richard A. Kemmerer. *Verification Assessment Study Final Report.* National Computer Security Center, Fort Meade, Maryland, 1986.

[14] Peter A. Lindsay. "A Survey of Mechanical Support for Formal Reasoning." Software Engineering Journal, January, 1988.

[15] Per Martin-Löf. "Constructive mathematics and computer programming." In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, North Holland, Amsterdam, 1982.

[16] J. McCarthy et al. *LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, Massachusetts, 1965.

[17] James T. Sasaki. *The Extraction and Optimization of Programs from Constructive Proofs.* PhD thesis, Cornell University, 1985.

Girard Normalization Proof in LEGO

Stefano Berardi

Dipartimento di Informatica
Torino, ITALY

Abstract

This paper is a report of the first attempt, in part successful, to formalize in LEGO (a mechanical proof - checker based on Pure Construction Calculus PCC) a large and difficult proof : Girard's proof of strong normalization for second order lambda calculus. We describe in an informal and general way the problems we had to face, since we believe that most of them are common to all proof - checkers implemented up to now.

Introduction: why is the formalization of Girard proof an interesting case of study

Girard's proof was choosen as the first sample of large proof to be formalized in LEGO. There are still very few examples of formalizations of large proofs in proof-checkers. Therefore they are very interesting, since they show problems which do not arise during short examples and tests. Actually, Girard's proof in its informal version is a difficult proof but it is not large. It becomes large since, in order to formalize it, we have to fulfill the task of proving every elementary step of it, in particular those concerning syntactical properties of second order lambda calculus. It becomes even larger because when we try to check it starting *from nothing*. We do not take for granted any set-theoretic or number-theoretic definition or property, not even those concerning logical connectives and rules! We assume only two very general axioms, *Infinity* and *Induction*. We had three more reasons to formalize Girard's proof. In a near future, we should be able to use LEGO, and all other proof-checkers, to check proofs concerning reliability and efficiency of programs. Girard's proof deals with syntactic properties of expressions of a formal language, a related topic. For this reason, formalizing Girard proof was also a good experience of formalization of a syntax (lambda calculus, or any functional language) inside a metalanguage (PCC, in our case), where the properties of the syntax can be stated and proved. Formalization of Elementary Mathematics is a must before any wide use of proof-checkers. It is better to organize it for a specified goal, than to do it in a non-systematic way with no aim at all. It turns out that to formalize the syntax required by Girard's proof, we have to define and check in LEGO most of Elementary Mathematics. (The formalization of syntax needs syntactic trees, booleans, integers, basic properties on them, basic notions on equivalence and order relations Ξ). Girard's proof, however, is not completely out of reach since it needs the notion of subset only in a few places. In section 4.1, we will explain why the notion of subset is so difficult to use in an intuitionistic proof. Essentially, in a constructive proof we know that a is in some subset A' only if we have a proof p of it. If f is some function on A', and we apply it to a, we have also to build a proof p' that (fa) is still in A'. Thus, we

[1]This work was partly supported by ESPRIT Basic Research Action "Logical Framework".

double the work concerning any operation on a subset. This is too bad for our understanding of the proof. In section 1, we start with some preliminaries, concerning PCC and the main features of the proof - checker LEGO. The reader may skip it, since the paper is more concerned with the pratical problems of formalizing proofs. In section 2, we briefly introduce the main problems we had in the formalization work. For each problem, we try to figure out how general it is. In section 3, we describe the first half of the formalization work, concerning (the general mathematical preliminaries to Girard's proof) Logic, Set Theory, Arithmetic. We briefly discuss the issue of the reliability of PCC itself, in view of some recent nonconservativity results [1]. In section 4, we describe the second half of the formalization work, concerning an introduction to the second order lambda calculus, the language of Girard's proof, and finally the lemmas and the main steps of Girard's proof.

1 Preliminaries on Pure Construction Calculus and LEGO

Pure Construction Calculus PCC was introduced five years ago by Coquand and Huet [2], expressly to be used as mechanical proof-checker. PCC is a system to derive jugments of the form $a : A$ (to be read as *a belongs to A*), where A is either a proposition (the formalization of a mathematical statement, or a small class) or a type (a larger class of objects). The class *Prop* of all propositions is a type (it belongs to the class *Type* of all types). *Type* is too large to be a class. A statement A is identified with the class of its proofs, and therefore it is considered as a (small) class. By the previous identification, we prove in PCC a proposition A if and only if we build an element a such that $a : A$. PCC is done in such a way that the relation $a : A$ (on pairs of expressions of PCC) is decidable. We can write down, once for all, a program to check it. Thus, if we already formalized in PCC a statement A with some supposed proof a , we can mechanically check if a is really a proof for A. The reader has surely noticed that the hard part of the work is to formalize A and a, or, better, to design a program in order to make such task easy. The main construction of PCC is (in LEGO syntax) the generalized cartesian product $\{x : A\}B$, where A and B are classes (i.e., propositions or types), and only B depends on the variable x. The class $\{x : A\}B$ consists of all functions f which send an element a of A in the element (fa) of $[x = a : A]B$ (of B with x replaced by a). Let b be element of B depending on the same variable x as B. Then the typical element of $\{x : A\}B$ is $[x : A]b$, to be considered as the function which sends an element a of A in the element $[x = a : A]b$ of $[x = a : A]B$. Remark that with are forced to identify the application $([x : A]ba)$, of the function $[x : A]b$ to a, with its result $[x = a : A]b$. The equivalence relation generated by such identification (on the expressions of PCC) is called beta-conversion. An important subcase is when B is a proposition. Then $\{x : A\}B$ is considered as the statement: *$[x = a : A]B$ holds, for all a in the class A.* In fact, thinking constructively, a proof f of *$[x = a : A]B$ holds, for all a in A* is nothing but a uniform way f to build, from any element a of A, a proof (fa) of $[x = a : A]B$. Therefore, f is nothing but an element of the generalized cartesian product $\{x : A\}B$. Suppose moreover A be a proposition and $[x = a : A]B$ be B (in other word, that B does not depend on x). In such subsubcase, the proposition $\{x : A\}B$ is written $A \to B$ and it is considered as the statement: *B holds, if A holds.* In fact, thinking constructively, a proof f of *B holds, if A holds* is nothing but a uniform way to build, from a proof a of A, a proof (fa) of B. Therefore f is nothing but an element of the function space $A \to B$. The last two interpretations may be in conflict, in other word we may think of an $A : Prop$ both as a domain of a quantification (as a small class) and as a statement. Such ambiguity has several advantages. We have in PCC the small class of integers, of booleans, of lists, ... Since a small class is also a statement, we may deal with an object in a small class using the same facilities we have for proofs. For instance, we have polymorphic programming (i.e., function with variable domain, corresponding to proofs of variable statements). However, as pointed out in Berardi [1] and in Geuvers [5], some pathological theorems may be derived in PCC using it. Thus, some small change is perhaps needed in PCC in order to avoid such pathologies, while leaving, say, polymorphic programming allowed. All the global variables are declared in a list called *the current context.* We start with the current context empty. If A is a proposition or a type, we may assume in PCC *x is an element of A*. Such assumption is formalized appending :

$$[x : A]$$

to the current context; If $[x : A]$ is the last assumption in the current context, and we want it to hold only in a given expression M, we deplace it to M:

$$[x : A]M.$$

If a is an expression inside some proposition or type A, we may assume *x is the name of a.* Such assumption is formalized appending :

$$[x = a : A](\text{or simply}[x = a])$$

to the current context. If $[x = a : A]$ is the last assumption in the current context, and we want it to hold only in a given expression M, we move it close to M :

$$[x = a : A]M.$$

Outline of LEGO

LEGO is a proof-editor for a SUN3 (or 4) workstation, supporting PCC and several related lambda-calculi. It is written in ML, a typed lambda calculus with implicit polymorphism. LEGO checks the correctness of an expression written in PCC syntax. Moreover, we may declare in LEGO a proposition or type A as goal. In other word, we declare we want to find some element a of A (say, A is a statement and we want to find a proof a of it). The editor LEGO has two basic tactics to help us building such an a : the *Intros* command and the *Refine* command Suppose A be

$$[x_1 : B_1] \cdots [x_n : B_n]C$$

in other words that a be a function sending $x_1, \ldots \wedge , x_n$ in some $(a\, x_1 \cdots x_n) : C$. We may ask, using the command :

$$Intros\ x_1 \ldots x_n;$$

to look instead for some $a' : C$, depending on extra free variables $x_1 : B_1, \ldots, x_n : B_n$. If we succeed, LEGO will automatically append to the current context :

$$[a = [x_1 : B_1] \cdots [x_n : B_n]a' : A].$$

Suppose now that

$$M : [x_1 : B_1] \cdots [x_n : B_n]C$$

be an expression of PCC. If we use the command :

$$Refine\, M;$$

LEGO adds to the current context the assumptions :

$$?1 : B_1, \ldots, ?n : B_n.$$

Then LEGO tries to find some $b_1 : B_1, \ldots, b_n : B_n$ such that

$$(M\, b_1 \cdots b_n) : A.$$

By definition of $[x_1 : B_1] \cdots [x_n : B_n]C$, we have

$$(M\, b_1 \cdots b_n) : [x_1 = b_1] \cdots [x_n = b_n]C.$$

Thus, LEGO has to find, using ?1, Ξ , ?n, some $[x_1 = b_1] \wedge [x_n = b_n]C$ equal (beta-convertible) to A. The research uses first order unification and beta-conversion. If LEGO succeds, we must supply a value for each variable among ?1, ... , ?n was effectively used. In other words, some B_i become new goals. If and when such new goals will be all solved, LEGO will automatically remove them form the current context, replacing each ?i by its value. Then, LEGO will append :

$$[a = (M\, b_1 \cdots b_n) : A]$$

to the current context. LEGO has several other commands and features to help proof- finding and context manipulation. The most striking one is an ML - style implicit polymorphism; but we skip it, since we would go too far from the topic of the paper. The reader should simply consider $[a|A]$, $\{a|A\}$ and $M \mid a$ as synonimous, respectively, of $[a : A]$, $\{a : A\}$ and $(M\, a)$.

2 The formalization of proofs

We may think of Girard's proof as an iceberg. In the top of it, we find what we usually consider the real proof; underwater, the most of the matter, consisting of all mathematical preliminaries a reader must know in order to understand what is going on. Such preliminaries are ranged from the general to the particular, and each of them is defined using, possibly, all the previous ones. Corresponding, we splitted the matter in six computer files :*Higher Order Logic and Elementar Set Theory*, *Peano Arithmetic*, *NonConservativity*, *The lambda calculus, the system F, a glossary for the Validity Theorem*, *A list of assumptions for Girard's Theorem*, *Girard's Theorem.* In the first three files, we find general preliminaries on mathematics. The three remaining files are more specific. Some problems concerning computer-aided proofs. The following problem concerning computer-aided proofs will arise, while describing the six previous files. They are ranged here from general to particular.

1. The Validation problem : what did we prove ? Even if we succeed in proving a formal goal, we still do not know if such formal goal is a correct translation of what we wanted to prove. Already for very short formulas some small doubt might arise. For instance, we might wonder if *ex2* $[obj, obj' : A]\{x : A\}(or(x = obj)(x = obj'))$ means exactly that A has cardinality at most two (in fact, it means that A has cardinality one or two, but not zero). In 4.2, we will give a more serious example, concerning very long and complex formulas. Such a problem holds for every computer - aided proof.

2. The Formalization problem for tactics : may we keep record of the method we used ? After we proved some goal, we might desire to keep record the method we used, in order to use it again. This is possible in a formal language enough flexible, where both mathematical statements and methods to deal with them are formal expressions with equal right. Such a problem holds for every purely formal system.

3. Subset representation. How to deal constructively with the subset notion ? The only constructive way known to deal with the subset notion is to consider an element a, of a subset A' of A, as a pair $\langle a, p\rangle$. The pair consisting of $a : A$ and of a proof p that a belongs to A'. This is very unpractical, since it force the length of a proof to double several times. Such problemwas already noticed during the Automath project, by de Bruijn's group. It holds for every constructive formal system.

4. How to deal with elementary mathematics : to assume or to prove it ? We may either consider as a new axiom any elementary fact we need in a proof (in such a case, we need only first order logic), or try to prove it using higher order logic. In the first case, life is easier but the constructive content of the proof, and its certainty maybe lost. Such a problem holds for every higher - order constructive formal system.

5. Conservativity problem : is our proof valid? The logic used by our proof checker could be different from the usual one. Are we really sure that every statement provable in proof checker logic is true in the usual sense ? Such problem holds for every higher - order constructive formal system which mix statements and small classes.

3 Basic Arithmetic and Logic

3.1 The file "Higher Order Logic and Elementar Set Theory"

The definitions in the first part of this file are the standard higher order definitions for *true*, *false*, *not*, *and*, *or*, *exists*, ... For instance, *and* is defined as :

$$[\mathit{and} = [A, B : \mathit{Prop}]\{X : \mathit{Prop}\}(A \to B \to X) \to X];$$

In other words, $(\mathit{and}\,A\,B)$ means that, for every proposition X, if A and B together imply X, then X holds. The most interesting fact is that the usual rules for those connectives becomes expressions of PCC, elements of some types. For instance, the rule *and-elimination-left*, which says *if (and A B) holds, then A holds*, becomes an element of $(\mathit{and}\,A\,B) \to A$ in the context $[A, B : \mathit{Prop}]$. We introduced in the file some new connectives and rules for purely practical reasons. For instance we introduced the connective *and3*, which takes three propositions A, B, C and says *A, B, C hold.* We introduced also the rule *ex-e2*, which says *if* $(P\,a)$ *and* $(Q\,b)$ *hold, for some a and b, and for every a and b we have* $(P\,a) \to (Q\,b) \to C$*, then C holds.* Such connectives, theorically, may be replaced by the usual ones. Anyway, they are useful, since they are closer to the way we effectively speak. In the second part of the file we introduced several elementar mathematical concepts, concerning basic operations on sets, equality and related notions, equivalence relations, isomorphisms, ... For instance, we define equality as Leibnitz equality. We consider a and a' equal elements of A if and only if they satisfy the same predicates, in other words if $(X\,a)$ is equivalent to $(X\,a')$ for every $X : A \to \mathit{Prop}$. Then we define the notion of functional, injective, ... all usual mathematical notions which require equality. The file is contains no new primitive concepts neither axioms. Higher - order logic allows us to replace each primitive concepts by a definition, and each axiom by a theorem.

3.2 The file "Peano Arithmetic"

Everyone accustomed with proof theory knows that the tricky part of a proof consists of a foule of arithmetical lemmas. This file, therefore, could be defined as the hidden *hardware* of Girard's proof. First of all, we define the small class Int of integers as :

$$[Int = \{C : Int\}(C \to C) \to C \to C : Prop];$$

Int is the intersections of all class of functional on C (functions on functions on C). It contains the iteration functionals :

- *zero* characterized by $(zero\ C\ fx) = x$,
- *one* characterized by $(one\ C\ fx) = (fx)$,
- *two* characterized by $(two\ C\ fx) = (f(fx))$,

Such functionals are identified with the natural numbers. Then we define several related notions, like the functions successor, predecessor, addition, product, power, the relations less, less or equal, and their main properties. They are outstandingly difficult to prove! For instance, consider the proof that the order relation *Less* is irreflexive :

```
irriflexivity_Less ==
[P=[n:Int]not (Less n n)]
induction P

  (nothing_less_zero zero
  : P zero)

  ([x:Int][induction_hyp:not (Less x x)][hyp:Less (S x) (S x)]
    induction_hyp
      (or_e
        (less_characterization_l hyp)
        ([hyp1:Less (S x) x]
          less_trans
            (less_succ_rule x)
            hyp1)
        ([hyp1:eq (S x) x]
          less_eq_trans
            (less_succ_rule x)
            hyp1)
       : Less x x)
    : {x:Int}(P x)->P (S x))
: Irriflexivity Less;
```

We assume induction as axiom, and from it we prove several related induction rules. Each function is defined as a polymorphic program, in other words as some $f : Int \to Int$ (or $f' : Int \to Int \to Int$, $\ldots$). Such possibility, typical of PCC and a few other languages, turns out to be very pleasant. Remind how tricky the standard logical way to introduce a function is ! First, we have to define the graph $F : Int \to Int \to Prop$ of f, then to introduce a new symbol f together with the definition axiom :

$$\{x, y : Int\}((fx) = y \leftrightarrow (Fxy))$$

The file contains only two assumptions : the fact that some small class has at least two elements, and the induction rule. It is unpleasant to have some assumptions forever in the current context, since, in general, we may extract the constructive content of a proof only if the context is empty. We cannot avoid to assume *some small class has at least two elements.* We cannot prove it in PCC, since it is consistent to assume in PCC that every small class has at most one element (for instance, we may consistently assume in PCC that zero is equal to one !). Induction, instead, is theorically useless. We might define N as the smallest subset of *Int* containing zero and closed by successor; then we might replace each quantification $\{x : Int\}(Px)$ by $\{x : Int\}(Nx) \to (Px)$. In such a way, essentially, we speak of $N = \{zero, one, two, \ldots\}$ instead of *Int.* Induction now holds, since it holds on N. This solution, however, is unpractical. We find here an example of the subset problem : if we replace everywhere $\{x : Int\}(Px)$ by $\{x : Int\}(Nx) \to (Px)$, then if we want to deduce $(P zero)$ from $\{x : Int\}(Nx) \to (Px)$ we need a proof of $(N zero)$. In other words, an integer n becomes in fact a pair $\langle n, p\rangle$, where p is a proof of (Nn). This almost doubles the length of a proof; for instance, if we defined a function $add : Int \to Int \to Int$, to use it we need also a proof of *if x, y are in N, then also (add x y) is in N.* Thus we were obliged to add the assumption $\{x : Int\}(Nx)$. This is not too bad from a constructive point of view. We may still extract the constructive content from the untyped version of the proof, using some trick inspired by model theory (we skip all details). The assumption $\{x : Int\}(Nx)$, however, has also a very unexpected drawback: we cannot assume Extensionality without inconsistency (see next file). Another problem arise in this file : handling inductive definitions, and more in general schemata of proofs. We inductively defined the order relation *Less* between integers as the smallest transitive relation $R : Int \to Int \to Prop$ such that $(Rx(Sx))$. We had to prove, then, that *Less* is transitive and that $(Less\ x(Sx))$. This is a general property : each set inductively defined satisfies each inductive clause. Other general properties hold; for instance, each element of the inductive set is obtained using a finite number of times the inductives clauses (if each inductive clause has a finite number of premisses). In LEGO, we have to repeat the proofs of such properties for each inductive definition. Christine Paulin (see [6]) put, in another proof-editor based on PCC, a command which returns, for each inductive definition, the proofs of those basic properties. This solution is the best one, up to now. If we extend such solution to each new schema of proof we need, however, the proof-editor could become difficult to use and not completley free of mistakes. Another solution could be to formalize in some variant of PCC (say, the general Construction Calculus XCC) the proofs of general properties of inductive sets, once for all. In general, we could try to formalize any new schema of proofs inside our formal system. In such a way, at least, we could check it mechanically and be sure it is completely free of mistakes.

3.3 The file "NonConservativity"

We already stated, introducing the file *Peano Arithmetic*, a drawback of the failure of Extensionality. Extensionality is a very useful axiom of Set Theory; it says that two sets with the same elements are the same set. In symbols, for $A : Prop$ and $X, Y : A \rightarrow Prop$:

$$(\{a : A\}(Xa) \leftrightarrow (Ya)) \rightarrow (X = Y).$$

We may consider a statement as a constant predicate on A and therefore as a set. Thus, extensionality implies that two equivalents propositions are the same, in symbols, if $B, C : Prop : (B \leftrightarrow C) \rightarrow B = C$. Unfortunatly, in PCC statements and small classes are both propositions; thus the innocuous theorem above can be also read : *if B and C are small classes, and there is a function $f : B \rightarrow C$, and a function $g : C \rightarrow B$, then B=C.* This theorem is crazy; we are now paying the price of the confusion between statements and small classes. In a previous paper (see [1]) I showed that, assuming Extensionality, we may prove that each inhabited small class is a lambda model. Then a contradiction with induction is easily derived:
by Tarsky fixed point theorem, every function on *Int* has a fixed point; by induction, the successor function has no fixed points (for no x we have $x = x + 1$).
Hence, in PCC we may prove *(not (and Extensionality Induction)).* This statement is false in *usual* mathematics. Technically speaking, PCC is nonconservative over *usual* mathematics. The whole proof outlined above is formalized and checked in the *NonConservativity* file. Another example of theorem of PCC, false in usual mathematics, is formalized and checked by a Philips Reasercher in [5]. A proof-checker using an inconsistent theory is not very reliable; thus, we must drop out either Extensionality or Induction. We choosed to drop out Extensionality. Also to drop out Extensionality has some drawbacks. For instance, suppose we checked that a given set $X$of lambda terms is a Girard's candidate. Then, if the set Y has the same elements of X, we must check again that Y is a Girard's candidate, since we do not know that $X = Y$. In other words, some one-step proofs have to be replaced by innaturally long reasonings. To avoid all the above pathologies, we should change a bit the definition of PCC in order to keep statements and propositions disjont. However, we should also keep some of the advantages of PCC, like polymorphic programming. Similar problems could hold for the systems related to PCC, like XCC (General Construction Calculus) and ECC (Extended Construction Calculus).

4 Girard's proof

4.1 The file "The lambda calculus, the system F, a glossary for the Validity Theorem"

This file prepares the language used during Girard's proof of normalization. First of all, we define the notion of lambda term in De Bruijn notion :

$$[Lterms = \{C : Prop\}(Int \rightarrow C) \rightarrow (C \rightarrow C) \rightarrow (C \rightarrow C \rightarrow C) \rightarrow C];$$

In other words, the set of lambda terms is the smallest set closed with respect a constructor *var* of variables, a unary operator *lambda*, a binary operator *appl.* Each lambda term obtained in this way lives in some context: (*var i*) lives in a some context including *(var zero), ..., (var i)*; (*lambda t*) lives in the context of t deprived of its last variable (bounded by *lambda*); (*appl t u*) lives in the same context as t, u. Then we define the basic operator on them, *alpha* (for context shifting) *sub* and *multisub* (for substitution) Ξ Those operator are introduced as polymorphic programs, as the operators on Int in the file *Peano Arithmetic.* In the same way we introduce second order types and related definitions. The notion of reduction and of of term typable in the second order lambda calculus are introduced as inductive definitions. The remaining part of the file is dedicated to the definition of Girard's candidate and candidate for a second order type. Finally we may state (not prove !) Girard's theorem together with its main lemmas. Sometimes, we require the notion of subset. For instance, when we define what untyped lambda terms are. Lambda terms are not the set of all formal expressions of the appropriate language, but the subset consisting of the sound expressions. The file is surprinsingly long and complex. Even if we glance at it, a question naturally arises: is the expression *GirardTheorem* : *Prop*, which we find at the end of the file, a correct translation of the informal Girard's Theorem? The doubt is not rethoric. Consider the definition of *GirardTheorem* :

```
GirardValidityTheorem ==
{ctxt:ListTypeVar}{G:TypedContext}{t:Lterms}{A:type}
                         {I:Interpretation}{i:interpretation}{C:UntypedContext}
                           [t' = (multisub (erasure G) t C i)]

                         (has_type_in_F ctxt G t A) ->

                         ({a:Variable}
                           (Belongs a ctxt) ->
                           Candidate (I a)) ->

                          ({x:Variable}
                          [B' = CandidateOf ctxt (extract_type G x) I]
                            (Belongs x (erasure G)) ->
                            (B' C (i x))) ->

                           [A' = CandidateOf ctxt A I]
                         A' C t';
GirardCorollary == GirardValidityTheorem -> GirardResult;
```

Each name used here was defined previously, sometimes with a definition even more complex, and required a chain of other definitions. How can be absolutely sure that, in each of those lines, we always faithfully translated what we had in mind? (of course, I strongly believe that we did). For instance, in a first version of the file, we typed in a definition *zero* instead of *one*. LEGO detected no mistakes, since we simply replaced a definition with another one; however, in such version of the file, the expression GirardTheorem did not mean *Girard Theorem* at all. I called this problem the *Validation Problem*, by analogy with the Validation problem for the specification of a program. The Validation problem holds also for polymorphic programs : how can we be absolutely sure that a program that we called *add* does really addition? In the case of a polymorphic program f, however, we may write and prove specification for f in PCC syntax, inside LEGO. For instance, we may write and prove in PCC the statements:

$$\{x : Int\}(add\ x\ zero) = x$$

$$\{x, y : Int\}(add\ x\ (Sy)) = (S(add\ x\ y))$$

and then check the proof in LEGO. In such a way, we become absolutely sure that *add* does addition. Using some generalization of those ideas to inductive definitions, we should be able to check that the the meaning of the PCC expression *GirardTheorem* is indeed Girard's Theorem. However, this is not yet be done. Some new ideas would be very welcome here.

4.2 The file "A list of assumptions for Girard's Theorem"

The file contain the work still to be done. It is a list of 20 assumptions, needed for Girard's proof and not yet proved. They are usually short syntactic or set-theoretical lemmas. All of them were used without proof in the original informal theorem. Except for some of them (for instance, the Substitution Lemma), we are in doubt if we should prove them or not. As a sample, we proved: *every variable strongly normalizes.* This stupid remark (obvious from the informal idea of normalization) required, as proof, a PCC expression several hundred chararacters long. More in general, we could wonder if the use of higher order logic, like in PCC, is worthwhile or not. Higher order logic (together with the Infinity axiom) is not easy to learn and difficult to use. It was adopted since it allows a foundation of *usual* mathematics: at least in principle, we may prove every assumption we needed in a proof. If we used intuitionistic higher - order logic, we may also (at least in principle) extract the constructive content from a theorem (after we discharged every assumption of it). However, checking every trivial assumption could take too long, and could not add that much to the reliability of a proof. Moreover, the program we may extract from a proof could be, in most of cases, too slow to be useful. We might wonder if first order logic could be more practical. First order logic is easier to learn and use, and naturally lead us to take, for each proof, a different set of *obvious* lemmas as new axioms. Possibly, this dilemma is not serious. In practice, in a first time we use higher order logic as first order logic (in a predicative way), leaving several minor assumptions of our proof unchecked. Even in this case, higher-order logic allows a more flexible and natural language. In a second time, if and when we believe that the proof is enough important, we always may use higher-order facilities (impredicative deductions) to fulfill the gaps we left in a first time.

4.3 The file "Girard Theorem"

The file was built using all prevous files (but *NonConservativity*). It is a very standard formal translation of informal proof. However, it required weeks (!). We should take in account, however, that this was an experimental work. It is not very common in general (and for both LEGO and PCC was the first time) that a large proof be formalized in a mechanical proof checker. When these experiments will become common, we may hope that a new generation of proof editors will be designed, expressely to be used in large proofs.

References

[1] Berardi, S., *Type Dependence and Constructive Types.* Ph. D. Thesis, Dipartimento di Matematica, Universita' di Torino, 1989.

[2] Coquand, T., Huet, G., The Calculus of Constructions. *Information and Conputation*, 1988.

[3] Geuvers, H., *Type Systems for HIgher Order Logic.* Catholic University of Nijmegen, draft, 1990.

[4] Girard, J.Y., *Interpretation Functionelle et Elimination de Coupures de l'Arithmetic d'Ordre Superieur.* These d' Etat, Universite Paris VII, 1972.

[5] Helmink, L., A Theorem due to Berardi, File circulated at the BRA meeting, Antibe, 1990.

[6] Paulin-Mohring, C., *Extraction de Programs dans le Calcul des Construction*, These d' Etat, Universite Paris VII, 1989.

[7] Paulin-Mohring, C., *Inductive Definitions in the Calculus of Constructions.* Rapport Technique INRIA 110. 1989.

[8] Pollack, R., Universes and Cummulativity. Draft.

Turning Eureka Steps into Calculations in Automatic Program Synthesis

Alan Bundy, Alan Smaill and Jane Hesketh

University of Edinburgh, Scotland

Abstract

We describe a technique called *middle-out reasoning* for the control of search in automatic theorem proving. We illustrate its use in the domain of automatic program synthesis. Programs can be synthesised from proofs that their logical specifications are satisfiable. Each proof step is also a program construction step. Unfortunately, a naive use of this technique requires a human or computer to produce proof steps which provide the essential structure of the desired program. It is hard to see the justification for these steps at the time that they are made; the reason for them emerges only later in the proof. Such proof steps are often call 'eureka' steps. Middle-out reasoning enables these eureka steps to be produced, automatically, as a side effect of non-eureka steps.

1 Introduction

We describe a technique called *middle-out reasoning* for the control of search during automatic program synthesis. Computer program synthesis is conducted by our OYSTER program, Horn [3], which was based on the Cornell University, Nuprl system, Constable *et al* [2]. To synthesise a program one provides a logical specification describing a relation, $spec(inputs, output)$, between the inputs and outputs of the proposed program. OYSTER is then used as an interactive theorem prover to prove a conjecture of the form:

$$\forall inputs\ \exists output\ spec(inputs, output) \tag{1}$$

in a logic based on Martin-Löf Intuitionist Type Theory[1], Martin-Löf [5]. This logic is *constructive*, *i.e.* the proof that an *output* exists for any combination of *inputs* must also show how to *construct* the *output* from the *inputs*. This construction recipe can be extracted from the proof and turned into a computer program.

OYSTER proceeds by a process of backwards reasoning from the goal. Associated with each of its backwards proof steps is a program construction step. As the proof proceeds a functional/logic program, $prog(inputs)$, which is also expressed in the Type theory logic, is constructed. This program meets the specification:

$$\forall inputs.\ spec(inputs, prog(inputs))$$

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks" and partly by SERC grant GR/E/44598 and an SERC Senior Fellowship to the first author. We are grateful for discussions with the other members of the mathematical reasoning group in the Artificial Inteligence Department at Edinburgh. This paper appeared in the proceedings of the UK-IT-90 conference.

[1]However, in the interests of wider readability we have translated the Martin-Löf language into a more conventional notation in the examples given below.

There is a duality between proof steps and program construction steps, *e.g.* an inductive proof produces a recursive program. Since recursion is pervasive in functional/logic programs, we are particularly interested in proofs by mathematical induction. Such proofs present especially difficult problems of search control, namely in the choice of induction rule and induction variable(s).

To automate this process of program synthesis we have built a program CLAM, van Harmelen [6], which guides OYSTER in its search for a proof. CLAM contains a collection of *tactics*. These are computer programs which apply OYSTER rules, and hence direct its search. CLAM analyses the conjecture to be proved and uses AI planning techniques to construct a *proof plan*, which is a tactic especially designed for the current conjecture. Further details of the OYSTER-CLAM system can be found in Bundy *et al* [1].

In OYSTER's logic, a conjecture of form (1) is usually proved by eliminating its quantifiers at some stage, to form a goal of the form:

$$spec(inputs, prog(inputs)) \tag{2}$$

where $prog(inputs)$ is called the *witness* of the existentially quantified variable *output*. This rather defeats the object of the exercise. As can be seen in goal (2), it is necessary for the OYSTER user to *provide* the very program which OYSTER is supposed to be synthesising. Even if the proof is first divided into cases and the quantifiers eliminated separately in each case, as in §3, the combination of the separate witnesses still defines the program.

The rest of the proof constitutes a mere *verification* that this program meets the specification, rather than a *synthesis* of the program. This elimination of an existential quantifier is an example of a *eureka* step, *i.e.* a step whose justification is not apparent at the time it is made. It becomes evident later in the proof, of course, when the verification succeeds. Eureka steps present a problem for both human and computer theorem proving. In either case it is hard to see how they might be thought of, *e.g.* what tactic might be implemented to make eureka steps.

Another example of a eureka step is the choice of appropriate induction variables and induction rule of inference. These will determine the forms of recursion to be used by the program.

2 Program Synthesis by Theorem Proving

To illustrate the technique of program synthesis and the eureka steps it requires, we will show how a program, $factors(x)$, for factorizing a positive integer, x, into its prime factors, can be synthesised from a proof of the prime factorization theorem. The prime factorization theorem can be expressed in English as:

> "For all positive integers, x, there is a list of prime numbers, xl, such that x equals the product
>
> of the numbers in xl."

It can be represented in our Type Theory as the formula:

$$\forall x{:}posint\ \exists xl{:}list(prime)\ prod(xl) = x \tag{3}$$

where $X : T$ means X is an object of type T, $posint$ is the type of positive natural numbers, $prime$ is the type of prime numbers, and $list(T)$ is the type of lists of objects of type T, *e.g.* $xl : list(prime)$

means xl is a list of primes. The function $prod : list(posint) \mapsto posint$ takes a list of numbers and returns their product, *i.e.*

$$\begin{array}{rcl} prod(nil) & = & 1 \\ prod(hd :: tl) & = & hd \times prod(tl) \end{array}$$

where nil is the empty list and :: is the infix list constructor.

We can prove this theorem easily by using the $prime\times$ induction rule:

$$\frac{\vdash_\alpha P(1) \qquad p\!:\!primes, x'\!:\!posint, P(x') \vdash_\beta P(p \times x')}{\vdash_\gamma \forall x\!:\!posint.\ P(x)}$$

i.e. we prove the theorem for $x = 1$, then we assume it for $x = x'$ and prove it for $x = p \times x'$, where p is a prime number. The greek letters labelling the $\vdash$ symbols are the program fragments associated with the proofs of these sequents. The program construction step associated with this induction rule defines the γ program in terms of the α and β program fragments, as follows:

$$\begin{array}{l} \gamma(x) \equiv \text{if} \quad x = 1 \quad \text{then} \quad \alpha \\ \qquad\quad\ \text{else } \beta(p, x') \text{ where } p = first(x) \wedge x' = rest(x) \end{array}$$

where $first(x)$ is the smallest prime number that divides x and $rest(x)$ is the quotient when x is divided by $first(x)$. In this case $factors(x) = \gamma(x)$. So the decision to use $prime\times$ induction ensures that the program will use a dual form of recursion. Deciding to use this esoteric form of induction is our first eureka step.

The base case of the induction is:

$$\vdash_\alpha \quad \exists xl\!:\!list(prime)\ prod(xl) = 1 \tag{4}$$

and the step case is:

$$\begin{array}{l} p\!:\!prime, \\ x'\!:\!posint, \\ \exists xl\!:\!list(prime)\ prod(xl) = x' \quad \vdash_\beta \\ \qquad \exists xl\!:\!list(prime)\ prod(xl) = p \times x' \end{array} \tag{5}$$

The base case is readily proved by eliminating the existential quantifier in (4) and introducing the witness nil for xl. This also instantiates α to nil. Similarly, the step case is proved by first eliminating the existential quantifier in the induction hypothesis part of (5) with witness xl' and then the one in the induction conclusion with witness $p :: xl'$. OYSTER is able to work out that xl' is $factors(x')$, so it instantiates β to $p :: factors(x')$. The program is now:

$$\begin{array}{l} factors(x) \ = \\ \qquad \text{if} \quad x = 1 \ \text{then } nil \\ \qquad \text{else } p :: factors(x') \\ \qquad\qquad \text{where } p = first(x) \wedge x' = rest(x) \end{array}$$

The program is now fully synthesised.

The witness introduced by the elimination of the existential quantifier in the induction hypothesis is standard, but the witnesses introduced by the elimination of the other two quantifiers constitute eureka steps. Note how these two witnesses are the values of the function $factors(x)$ in the base and step cases of the recursion.

3 The *Ripple_Out* Tactic

The remaining parts of the proof will provide a verification that our choices of induction rule and existential witnesses yield a program that meets the specification. The remaining part of the step case is to prove:

$$
\begin{array}{rcl}
p:prime, & & \\
x':posint, & & \\
xl':list(prime) & & \\
prod(xl') = x' & \vdash_\beta & \boxed{p ::} xl':list(prime) \wedge \\
 & & prod(\boxed{p ::} xl') = \boxed{p\times} x'
\end{array}
\tag{6}
$$

Note that the existential quantifiers have all been eliminated and the existential variables replaced by their witnesses. The witness of the existential variable in the induction conclusion, $p :: xl'$ must be shown to have the right type, $list(prime)$, so the subgoal, $p :: xl':list(prime)$, becomes part of the induction conclusion.

Three sub-expressions of the induction conclusion have been placed in boxes. These are examples of *wave fronts*. *Wave fronts* are those sub-expressions of the induction conclusion in which it differs from the induction hypothesis. The CLAM system contains a tactic called *ripple_out* whose task is to make the induction hypothesis appear as a sub-expression of the induction conclusion. It works by rewriting the induction conclusion to move the wave fronts outwards from their original deeply nested positions. The rules used by *ripple_out* are called *wave rules*. Wave rules are rewrite rules of the form[2]:

$$F(\boxed{S_1(\boxed{U_1})},\ldots,\boxed{S_n(\boxed{U_n})}) \quad \Rightarrow \quad \boxed{T(\boxed{F(U_1,\ldots,U_n)})}$$

where F, T and the S_i are terms with distinguished arguments and T may be empty, but F and the S_i must not be. The S_i are old wave fronts and T is the new wave front. Application of a wave rule ripples some wave fronts out by one stage. Repeated application ripples them all to the outside of the induction conclusion.

The wave rules required for this proof are:

$$\boxed{hd ::} tl:list(type) \quad \Rightarrow \quad \boxed{hd:type \wedge} tl:list(type) \tag{7}$$

$$prod(\boxed{hd ::} tl) \quad \Rightarrow \quad \boxed{hd\times} prod(tl) \tag{8}$$

$$\boxed{u\times} v = \boxed{u\times} w \quad \Rightarrow \quad v = w \tag{9}$$

[2]This is not the most general form that wave rules can take, but is sufficient for the examples in this paper. A more general form is given in Bundy *et al* [2].

Applying these wave rules to the induction conclusion of (6) rewrites the step case as follows. After application of rule 7 to the first wave front, CLAM derives:

$$\begin{array}{rl}
p\!:\!prime, & \\
x'\!:\!posint, & \\
xl'\!:\!list(prime) & \\
prod(xl') = x' & \vdash_\beta \quad \boxed{p : prime \wedge}\, xl'\!:\!list(prime) \wedge \\
& \qquad prod(\boxed{p ::}\, xl') = \boxed{p \times}\, x'
\end{array}$$

After application of rule 8 to the second wave front, it derives:

$$\begin{array}{rl}
p\!:\!prime, & \\
x'\!:\!posint, & \\
xl'\!:\!list(prime) & \\
prod(xl') = x' & \vdash_\beta \quad \boxed{p : prime \wedge}\, xl'\!:\!list(prime) \wedge \\
& \qquad \boxed{p \times}\, prod(xl') = \boxed{p \times}\, x'
\end{array}$$

And after application of rule 9, simultaneously, to the second and third wave fronts, it derives:

$$\begin{array}{rl}
p\!:\!prime, & \\
x'\!:\!posint, & \\
xl'\!:\!list(prime) & \\
prod(xl') = x' & \vdash_\beta \quad \boxed{p : prime \wedge}\, xl'\!:\!list(prime) \wedge \\
& \qquad prod(xl') = x'
\end{array}$$

After these three rule applications each of the three conjuncts of the induction conclusion is identical to one of the induction hypotheses. The induction hypothesis can then be used to prove the induction conclusion. The CLAM system has a tactic called *fertilization* whose task is to match the induction hypothesis against sub-expressions of the induction conclusion and replace any such sub-expressions by *true*. Doing this completes the proof of the step case in this example.

A fuller explanation of *ripple_out* can be found in [2].

4 Middle-Out Reasoning

We can think about the eureka steps described in §2 in the following way. The choices of induction scheme and existential witnesses are actually made in such a way as to allow the rest of the proof to proceed successfully. In particular, the eureka choices will enable the subsequent *ripple_out* tactic to succeed. However, since the eureka steps are made before *ripple_out* is applied, this is not immediately obvious.

This observation suggests a way to automate the eureka decisions, namely: postpone making the eureka steps and apply *ripple_out first*; then integrate the eureka steps into the rippling as required to enable it to continue. Effectively we do the middle of the proof first. In the process we calculate what form the beginning of the proof should take to make the middle of it succeed. We call this strategy *middle-out reasoning*.

4.1 Rippling Under Existential Quantifiers

We consider first the problem of choosing witnesses for existentially quantified variables. Our solution is not to eliminate the existential quantifiers at all, but to modify the rewriting procedure so that rippling can be carried out under existential quantifiers. To illustrate this process, consider again the proof of the prime factorization theorem. We return to the step case of the proof, just after the application of induction.

$$\begin{array}{rl} p\!:\!prime, & \\ x'\!:\!posint, & \\ \exists xl'\!:\!list(prime)\ prod(xl') = x' & \vdash_\beta \ \ \exists \boxed{xl}\!:\!list(prime)\ prod(\boxed{xl}) = \boxed{p\times}x' \end{array} \tag{10}$$

This time we have marked in the wave fronts *before* eliminating the existential quantifier. Recall that the witness for xl in the induction conclusion, $p :: xl'$, contained a wave front, $\boxed{p ::}$, (see (6)). This justifies our marking this existential variable as a wave front.

This remark generalises to all existentially quantified variables in induction conclusion. Their witnesses will be the value of the synthesised program in the step case of the recursion, *i.e.* some function of the value of the program when it is called recursively. When this function is not the identity function, then it will be a wave front. So all existential variables in induction conclusions are potential wave fronts. Note that the occurrence of the existential variable in the quantifier declaration is marked as a wave front, as well as all occurrences in the body of the formula. This is because the quantifier declaration also contains a type declaration, which will require rippling with wave rules for types, like (7) above.

We now proceed to apply *ripple_out*, using the same wave rules as in §3, but without first removing the $\exists xl : list(prime)$. Our modified rewriting procedure has the freedom to instantiate existential variables to compound terms of the same type. This is because in the unmodified procedure these compound terms could have been introduced as witnesses before *ripple_out* was applied. To take advantage of this possibility the modified rewriting procedure matches the left hand side of the rewrite rule with expressions in the goal, treating existential variables as free variables that can be instantiated. When an existential variable is instantiated to a witness of the same type, the existential quantifier that governs it will no longer be required, and should be deleted. However, a new existential quantifier will be required for each variable contained in the witness. To assist with checking that the witness has the required type and with the calculation of the types of the new existential variables, it helps if the wave fronts in the quantifier/type declarations are rippled before those in the body of the formulae.

Consider, for instance, the application of wave rule (7) to the first wave front in (10).

$$\boxed{\exists hd\!:\!prime}\ \exists tl\!:\!list(prime)\ prod(\boxed{hd ::}tl) \quad = \quad \boxed{p\times}x'$$

This instantiates xl, and hence β, to $hd :: tl$. The existential quantifier governing xl is replaced by two existential quantifiers: one for hd and one for tl. Note how the types of these new existential variables are provided by the right hand side of rule (7). This use of a type rule also guarantees that the witness has the right type. Note also how the second occurrence of xl, in the body of the formula, has been instantiated to $hd :: tl$.

The witness to the existential variable, xl, has now been determined. It was not necessary to do this as a eureka step. The appropriate witness was calculated by the matching routine during the application of *ripple_out*. No search was required for this. The wave front around xl had to be rippled out at some time (in fact, it is best to ripple it out first). Rule (7) was the only matching wave rule because of the restriction imposed by the sub-expression $list(prime)$, the type of xl.

We now apply wave rule (8) to the second wave front, yielding:

$$\boxed{\exists hd{:}prime}\ \exists tl{:}list(prime)\ \boxed{hd\times}\,prod(tl) \quad = \quad \boxed{p\times}\,x'$$

This application does not instantiate any existential variables, so does not require any changes in the existential quantifiers.

We can now apply multi-wave rule (9), simultaneously, to the second and third wave fronts, yielding:

$$\exists tl{:}list(prime)\ prod(tl) \quad = \quad x'$$

This rewrite instantiates hd to p. This instantiation is only possible because p is a term with the same type as hd. Since hd has been instantiated its existential quantifier must be removed. Since p is a constant, and therefore contains no new variables, no new existential quantifiers are introduced. This has the side effect of removing the first wave front. The remaining formula matches the induction hypothesis. So fertilization removes it, instantiating tl to xl', and hence xl and β to $p :: xl'$, as required.

A similar process can be carried out in the base case. The role of *ripple_out* being played by the CLAM tactic, *base*, which rewrites formulae using the base cases of recursive definitions.

4.2 Using Meta-Variables for Induction Terms

We now consider the problem of choosing induction rules. Our solution is make a 'least commitment' induction step by using a schematic induction rule in which the induction term is a meta-variable. We then allow this meta-variable to become instantiated during the subsequent rippling out. Since meta-variables are not allowed in OYSTER's logic, this reasoning is handled within the CLAM planner. CLAM applies *ripple_out* and works out what induction rule is required. It then instructs OYSTER to apply the appropriate induction rule and then ripples the wave fronts that induction creates.

To illustrate this we return again to the step case of the prime factorization proof, just after the application of induction, *i.e.*

$$\begin{array}{c} x'{:}posint, \\ \exists xl{:}list(prime)\ prod(xl) = x' \end{array} \ \vdash_\beta \ \exists \boxed{xl}{:}list(prime)\ prod(\boxed{xl}) = \boxed{X}$$

but instead of the induction term, $p \times x'$, we have used the meta-variable, X, which stands for some term containing x', *i.e.* we have replaced all occurrences of x', in the induction conclusion, by X. All universally quantified variables are candidate induction variables, but x is the only candidate in this example. In general, we must try replacing each universal variable, in turn, by such a meta-variable, to see which replacements permit *ripple_out* to succeed. Note that the declaration of p has been omitted from the hypothesis, since we do not yet know what parameters might be introduced by the induction.

Rippling-out then proceeds as in §4.1 until we get to the formula:

$$\boxed{\exists hd{:}prime}\ \exists tl{:}list(prime)\ \boxed{hd\times}\,prod(tl) \quad = \quad \boxed{\mathrm{X}}$$

Now rule (9) is the only wave rule that applies to the second wave front. Applying it produces:

$$\boxed{\exists hd{:}prime}\ \exists tl{:}list(prime)\ prod(tl) \quad = \quad X'$$

where X is instantiated to $hd \times X'$. $fertilization$ applies to this, instantiating X' to x' and leaving the residue:

$$\exists hd{:}prime\ true$$

which is provable provided hd is witnessed by a prime number.

We have thus established that the *ripple_out* tactic will succeed provided that the induction term is $hd \times x'$, where $hd{:}prime$. Induction rules are indexed in CLAM by their induction terms, so this information enables the systems to recover the $prime\times$ induction rule and use it retrospectively.

Notice how the appropriate induction rule was chosen as a side-effect of the *ripple_out* tactic.

5 Conclusion

In this paper we have described a technique, based on theorem proving, for synthesising programs from logical specifications. We have seen that a naive use of this technique requires eureka steps. In §2 we saw that it was necessary to produce an appropriate induction rule and appropriate existential witnesses 'out of the blue'. Furthermore, these eureka steps provided all the essential structure of the synthesised program, leaving only the verification that this program met the specification. These eureka steps constitute a barrier to the use of the program synthesis technique. This is true whether we want to use it totally automatically, with a computer providing the eureka steps, or semi-automatically, with a human providing them.

We have described a way of finessing the eureka steps, so that program synthesis can be automated. We draw on previous work in which we automated the verification part of the proof. In particular, our *ripple_out* tactic is highly successful in automatically guiding the proving of the induction conclusion from the induction hypothesis. We have arranged the proof construction so that this middle part of the proof is done first. The eureka steps emerge as a side effect of *ripple_out* — they are made in such a way as will permit *ripple_out* to continue. We call this technique *middle-out reasoning*. We are currently implementing middle-out reasoning within the OYSTER-CLAM system.

References

[1] Bundy, A., van Harmelen, F., Hesketh, J. and Smaill, A., "Experiments with Proof Plans for Induction", Jour. Auto. Reas., 1990, in press. Earlier version available from Edinburgh as Research Paper No 413.

[2] Bundy, A., van Harmelen. F., Smaill, A., "Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs", Research Paper 459, Dept. of Artificial Intelligence, Edinburgh, 1990. To appear in the proceedings of CADE-10.

[3] Constable, R.L., Allen, S.F., Bromley,H.M., *et al*, "Implementing Mathematics with the Nuprl Proof Development System", Prentice Hall, 1986.

[4] Horn, C., "The Nurprl Proof Development System", Working Paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[5] Martin-Löf, Per, "Constructive Mathematics and Computer Programming", 6th International Congress for Logic, Methodology and Philosophy of Science", Hanover, pp153-175, August, 1979, Published by North Holland, Amsterdam. 1982.

[6] van Harmelen, F. "The CLAM Proof Planner, User Manual and Programmer Manual", Technical Paper 4, Dept. of Artificial Intelligence, Edinburgh, 1989.

A Natural Deduction treatment of Operational Semantics[1]

Rod Burstall[2] Furio Honsell[3]
University of Edinburgh University of Udine

June 15, 1990

Abstract

We show how Natural Deduction extended with two replacement operators can provide a framework for defining programming languages, a framework which is more expressive than the usual Operational Semantics presentation in that it permits hypothetical premises. This allows us to do without an explicit environment and store. Instead we use the hypothetical premises to make assumptions about the values of variables. We define the extended Natural Deduction logic using the Edinburgh Logical Framework.

1 Introduction

The Edinburgh Logical Framework (ELF) provides a formalism for defining Natural Deduction style logics [5]. Natural Deduction is rather more powerful than the notation which is commonly used to define programming languages in "inference-style" Operational Semantics, following Plotkin [12] and others, for example Kahn [6]. So one may ask

> "Can a Natural Deduction style be used with advantage to define programming languages?".

We show here that, with a slight extension, it can, and hence that the ELF can be used as a formal meta-language for defining programming languages. However ELF employs the "judgements as types" paradigm and takes the form of a typed lambda calculus with dependent types. We do not need all this power here, and in this paper we present a slight extension of Natural Deduction as a semantic notation for programming language definition. This extension can itself be defined in ELF.

The inspiration for using a meta-logic for Natural Deduction proofs comes from Martin-Löf. Our work benefited from that of Mason [8] who did a proof system for Hoare logic in the ELF, encountering problems about the treatment of program variables. In particular he adopted the non-interference relation originally used by Reynolds [3]. The main feature of Natural Deduction proofs, used in our semantics but not used in the usual style of Operational Semantics, is that the premises of a rule may be hypothetical: they may themselves be of the form "$\mathbf{Q}$ is derivable from $\mathbf{P}_1$ and ... and $\mathbf{P}_1$". We write a premise of this form thus

$$\frac{\begin{array}{c}(\mathbf{P}_1,\ldots,\mathbf{P}_n)\\ \vdots\\ \mathbf{Q}\end{array}}{\mathbf{R}}$$

[1]Work partially supported by the UK SERC, by the Italian MPI 40% and 60% grants, by the Project Stimulation Lambda Calcul Type' and by the Basic Research Action project: Logical Frameworks.

[2]Dept of Computer Science, Edinburgh University, JCMB, King's Bdgs., Mayfield Rd., Edinburgh EH9 3JZ, U.K.

[3]Dipartimento di Informatica, Universita di Udine, via Zanon 6, 33100- Udine, Italy

Using these techniques we are able to give an operational semantics which dispenses with the traditional notions of environment and store. This makes our semantic definitions and our proofs of evaluations appreciably simpler than the traditional ones. Our proofs are the same shape as the traditional ones, but each formula is simpler; the environment and store do not appear repeatedly in formulas as they do in the traditional system. We exploit in fact the structural rules inherent in Natural Deduction. Instead of environment and store we use the notions of expression with an attached substitution and evaluation of an expression after a command (compare Dynamic Logic). Instead of evaluating an expression M with respect to an environment we consider its value given some assumptions about the values of the variables which occur in it. We discuss also an alternative approach in which commands evaluate to "partial stores".

The main technical difficulty is to define the necessary substitution operations; these play a crucial roles in our semantics. For instance we need to define substitutions of values for identifiers in expressions, declarations and commands. This is not a textual substitution; it depends on the binding operators in the language being defined. For example the meaning of $[m/x]M$ (substitute value m for identifier x in expression M) is that when we come to evaluate M we assume the value of x to be m and ignore any previous assumptions about the value of x. To express this we use the hypothetical premises of Natural Deduction, with side conditions to ensure that we have a fresh variable. However we are not able to express substitution purely in a Natural Deduction logic. The difficulty can be reduced to substituting a (new) identifier for an (old) one. Thus we have to add a primitive operator for identifier replacement, just textual replacement, which we call α. (It is named from α conversion in lambda calculus.) We define this with a special rule schema. We also need an "dual" to a which we call $\underline{\alpha}$.

In using an editor or a programming language you probably learn from a "Users' Manual" and then, having got some experience, look up the fine points in a "Reference Manual". For our style of doing operational semantics the "Reference Manual" is a formal description in ELF. We try to provide also a less formal "Users' Manual" which describes the form of the semantic rules and the criteria for proofs utilizing a particular notation called alpha notation. Alpha notation will be described with about the degree of precision with which the rules of predicate logic might be defined in a text book. As usual we will give the Users' Manual first, asking the reader to suspend critical judgement while getting an intuitive feel for the style of semantics we propose.

We proceed in four steps.

- We define alpha notation: Natural Deduction extended with operators α and $\underline{\alpha}$ to replace variables.
- We use this notation to define the notion of an Evaluation System, by introducing evaluation predicates, $\Rightarrow$, and substitution operators, $[\ \ /\ \ ]$, obeying suitable rules.
- We use an evaluation system to define the semantic rules for a sample programming language which features lambda expressions, commands, exceptions, declarations, procedures and expressions with side effects.
- We give a formal definition of an Evaluation System in the Edinburgh Logical Framework.

In this last step we show the connection of our approach with the ELF treatment of logical languages. First we give a literal encoding of evaluation systems in ELF. We thus show how to put alpha notation on a firm foundation by explaining it in terms of higher order constructions. Finally we give an alternative encoding which does not mention alpha notation at all.

1.1 Comparison with the Edinburgh Logical Framework

Our original aim was to use the ELF as a definition medium for Programming Languages. We achieve this in the last step above. Our semantic rules can be written in ELF notation by mere transliteration. However the rules do not use all the power of the ELF except in defining the substitution operations. It seemed to us better to present Evaluation systems as a simpler framework for semantics. These do not need a higher order description of syntax and do not introduce involved judgements which use the dependent types structure of ELF. Instead they makes use of the primitive operators α and $\underline{\alpha}$. The semantic rules do not even mention α or $\underline{\alpha}$, once we have defined the appropriate substitution rules. In short:

> ELF approach: If you understand lambda calculus with dependent types you can define many logics formally.
>
> Evaluation systems approach: If you understand Natural Deduction with variable replacement you can define many programming languages formally.

In using evaluation systems we lose the ELF advantage of a built-in type checker for dependent types, so that we have to write a separate static semantics (not treated in this paper). Even in ELF we may have to write a static semantics, for example to handle phenomena such as the polymorphism in ML.

1.2 How this paper evolved

This work started while Furio Honsell was at Edinburgh University. An early version of this paper appeared as an invited paper in Proc. 8th Conf. on Foundations of Software Technology held in Pune (India) 1988 [1] and as an LFCS Report [2]. This is a revised and expanded version with some corrections. The main changes with respect to the earlier versions are (i) the rules for applying a closure or a primitive; (ii) the example proofs; (iii) removal of explicit equality in alpha notation; (iv) the rule of substitution uses $\underline{\alpha}$; (v) recursive function definition; (vi) rules for complex declarations; (vii) alternative rules for commands; (viii) expressions with side-effects; (ix) a concise ELF encoding which does not use α s nor $\underline{\alpha}$ s.

Since this first version of this paper appeared a lot of work has been carried out in the direction of representing operational semantics in a Logical Framework. In particular Hannan and Miller [9] discussed operational semantics for a purely functional language in λ-Prolog utilizing full-blown higher order syntax. C.-E. Ore [11] discussed a more efficient but perhaps less uniform encoding of our notion of Evaluation System in ELF and argued against the possibility of using Natural Deduction in reasoning about heaps.

1.3 Acknowledgements

We thank our colleagues at the Laboratory for Foundations of Computer Science for stimulating interaction. We would like to thank Randy Pollack who coded our semantics in his LEGO ELF [13] system and thereby brought to light two errors in our earlier version of the paper. Rod Burstall also thanks Butler Lampson of DEC SRC for experience gained in their joint work on the semantics of the Pebble language. Thanks to Claire Jones for kindly Texing this paper.

2 Definition of a Framework for Operational Semantics

Alpha notation is about syntactic entities, namely identifiers and expressions. As we pointed out in the introduction, the main reason for introducing it is that of simplifying our framework. In particular using alpha notation we succeed in doing without a higher order representation of syntax and complex judgement constructions.

2.1 Alpha notation

Let Σ be a first order signature with two sorts, *Id* (identifiers) and *Expr* (expressions), some predicates P and some function symbols F. We introduce two distinguished function symbols α, $\underline{\alpha}$: $Id \times Id \times Expr \to Expr$; α denotes replacement of all occurrences of an identifier in a term by another identifier, whilst $\underline{\alpha}$ is "dual" to it.

We define Terms and (atomic) Formulas over the signature and variables as usual. Rules are defined as in Natural Deduction.

Alpha notation over a signature Σ with sorts *Id* and *Expr* uses the distinguished function symbols α and $\underline{\alpha}$. These are subject to a set of rules $\mathbf{R} \cup \underline{\mathbf{R}}$ which we now define.

An element of $\mathbf{R}$ is a well formed instance of the following schema:

$$\frac{C[J]}{C[\alpha_{j/i} I]}$$

where $C[\ ]$ is any context (formula with a hole) and where J is the term obtained from the term I by replacing all occurrences of identifier i with j.

Thus to prove $\alpha_{j/i}(i+k) < 3$ we have to prove $j + k < 3$.

An element of $\underline{\mathbf{R}}$ is a well formed instance of the following schema:

$$\frac{C[J]}{C[\underline{\alpha}_{j/i} I]}$$

where $C[\ ]$ is any context (formula with a hole) and where I is the term obtained from the term J by replacing all occurrences of identifier j with i.

Thus if we have proved $i + j < 3$ we may conclude $\underline{\alpha}_{j/i}(i + i) < 3$. (We may reach the same conclusion from $i + i < 3$ or $j + i < 3$ or $j + j < 3$.) The reader may like to think of $\underline{\alpha}_{j/i}$ as dual to $\alpha_{i/j}$.

A proof in alpha notation is a natural deduction proof [14]. In explaining the set of rules $\mathbf{R} \cup \underline{\mathbf{R}}$ we appealed to the intuitive understanding of the phrase "replacing all occurrences of identifier j with i". In the appendix we will give a formal account of the notion of a proof in alpha notation in the Edinburgh Logical Framework. The main step will be to reflect in the system the notion of syntactic identity so as to make it possible to reason formally about what we use informally at this stage.

More generally a "many sorted alpha notation" has sorts $Id_{s_1}, \ldots, Id_{s_n}$ and $Expr_{t_1}, \ldots, Expr_{t_n}$, the function symbols, $\alpha_{st}, \underline{\alpha}_{st}$: $Id_s \times Id_s \times Expr_t \to Expr_t$ and corresponding sets of rules.

2.2 Evaluation systems—1

We now use alpha notation to define evaluation systems. We consider first the case with two sorts Id (identifiers) and Expr (expressions). An evaluation system in alpha notation is a formal system which contains, in addition to the function symbols and the rules of alpha notation, at least two distinguished predicates, a distinguished function and certain rules which these obey. An evaluation system will contain also functions, predicates and rules specific to the object language it defines.

The distinguished predicates correspond to the notions of evaluation and value denoted $\Rightarrow$ over $Expr \times Expr$ (written infix) and $Value$ over $Expr$ respectively.

The distinguished functions are $expr$ (to convert Id to $Expr$) and substitution; denoted $expr : Id \to Expr$ and $[\ \ /\ \ ]: Expr \times Id \times Expr \to Expr$ respectively.

For ease of reading we will usually omit the conversion function $expr$ hereafter, treating Id as a subsort of $Expr$.

We use the following symbols (possibly primed): x, y, z for identifiers; M, N, m, n, p for expressions.

There are two rules. The first rule is

$$\frac{value\ n \quad \begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ \alpha_{x'/x}M \Rightarrow \alpha_{x'/x}m\end{array}}{[n/x]M \Rightarrow m}\ x' \text{ is a new variable}$$

By "x' is a new variable" we mean that is it does not occur in n, x, M and m nor does it occur in any assumption except those of the form $x' \Rightarrow n$; such assumptions are discharged by an application of this rule.

This rule is really no more complex than the usual rule for, say, existential elimination. The reason for this simplicity is our appeal to the intuitive understanding of the alpha operators. The ELF encoding of this rule, that we will give in the appendix, will be much more convoluted. In establishing the minor premise, three more assumptions will be allowed, all of which will be discharged by an application of the rule itself. The reason for this is due to the fact that, x' is a new variable, has to be reflected in the system itself in order to apply the formal versions of the rules governing the alpha operators. We will give also an encoding of evaluation systems with no formal counterpart for the alpha operators. The encoding of the substitution rule will then be even more complex. Hence there is a case for alpha operators even in the ELF encoding. They factorize a complex rule into more elementary ones.

The second rule is

$$\frac{value\ n}{n \Rightarrow n}.$$

The rule for substitution is the key definition. It encapsulates the way we handle object language variables. One of the virtues of the Natural Deduction framework used in evaluation systems is that it provides uniformity in handling scopes of variables, avoiding side conditions on rules. This is important in practice because handling variables is a traditional pitfall in defining logics or programming languages. Our approach is to define substitution in terms of evaluation.

A naive version of the substitution rule might have been

$$\frac{value\ n \quad \begin{array}{c}(x \Rightarrow n)\\ \vdots \\ M \Rightarrow m\end{array}}{[n/x]M \Rightarrow m}$$

but the evaluation of M in the hypothetical premise could make use not only of $x \Rightarrow n$, but also of any other statement about the value of x in the context in which the rule is used; we might be evaluating $[n/x]M$ in a context in which $x \Rightarrow n'$ and this should not affect the value. Thus we need to introduce a new variable x' local to the hypothetical premise. We have to replace x by x' in M in the right subproof. Just in case the resulting value m in the subproof contains an x' we have to replace the x' by an x (This will occur in our example language only for closures).

2.3 Syntactic sugar

We will mostly use infix syntax for expressions, e.g. "**let** $x = M$ **in** N" for "**let**(x, M, N)". When we introduce a syntax for a particular object language in what follows this is to be understood as syntactic sugar.

2.4 Example proof

We now give a proof using alpha notation with the substitution rule above, constants 1,2,3, an operation + and three extra rules

$$\frac{M \Rightarrow 1 \quad N \Rightarrow 2}{M + N \Rightarrow 3} \qquad\qquad \frac{}{\mathit{value}\ 1} \qquad\qquad \frac{}{\mathit{value}\ 2}$$

We will show that $[1/x][2/y]x + y \Rightarrow 3$.

$$\frac{\mathit{value}\ 1 \qquad \dfrac{\dfrac{\mathit{value}\ 2 \qquad \dfrac{\dfrac{[x' \Rightarrow 1]_{(1)} \quad [y' \Rightarrow 2]_{(2)}}{x' + y' \Rightarrow 3}}{\alpha_{y'/y} x' + y \Rightarrow \underline{\alpha}_{y'/y} 3}}{[2/y]x' + y \Rightarrow 3}\ (2)\ y'}{\underline{\alpha}_{y'/y}[2/y]x + y \Rightarrow \underline{\alpha}_{x'/x} 3}}{[1/x][2/y]x + y \Rightarrow 3}\ (1)\ x'$$

The square brackets enclose hypotheses which are discharged at the level indicated by the subscript. The scope of a variable is shown by writing it at the end of the line below the scope.

We could write this proof more briefly omitting the applications of the α and $\underline{\alpha}$ rules. We may think of the α expression brought in by the substitution rule being immediately reduced. A proof editor could do this step automatically. We recommend this style of proof display. Introducing α and $\underline{\alpha}$ is a technical device for making the machinery of substitution explicit. They do not appear in the semantic rules once we have defined substitution, and they can well be omitted from the proofs.

$$\frac{\mathit{value}\ 1 \qquad \dfrac{\mathit{value}\ 2 \qquad \dfrac{[x' \Rightarrow 1]_{(1)} \quad [y' \Rightarrow 2]_{(2)}}{x' + y' \Rightarrow 3}}{[2/y]x' + y \Rightarrow 3}\ (2)\ y'}{[1/x][2/y]x + y \Rightarrow 3}\ (1)\ x'$$

2.5 Evaluation systems — 2, The multisorted case

So far we have treated the case with one sort *Id* and one sort *Expr*, in a many sorted case these would be indexed families Id_s and $Expr_t$. We may define an evaluation system over a multi-sorted alpha notation with sorts Id_s and $Expr_t$ and possibly with more than one evaluation predicate, value predicate and substitution function. Thus we have

$$\alpha_{ij}, \underline{\alpha}_{ij}: Id_i \times Id_i \times Expr_j \to Expr_j$$

$$expr_{ik}: Id_i \to Expr_k$$

$\Rightarrow_k$ over $Expr_k \times Expr_k$, $value_k$ over $Expr_k$ and $[\quad / \quad]_{ikm}: Expr_k \times Id_i \times Expr_m \to Expr_m$. Corresponding to each substitution function an appropriate substitution rule schema is added, or even several rule schemas corresponding to different evaluation predicates. These schemas will always be of a similar pattern, often just polymorphic variants of the single-sorted case. We will not give a precise definition of these schemata here. Examples of multi sorted evaluation systems with more than one substitution operation and evaluation predicate will be found in the semantics for local variables in commands and for complex declarations.

2.6 Conventions for syntax

To extend the evaluation system signature $(Id, Expr, \Rightarrow, value, [\quad / \quad])$ we will use the usual conventions of syntax. These allow us to introduce function symbols together with the infix notation for them. We will use the names allocated to schematic variables for the syntax classes. For example using x, y, z for identifiers; M, N, m, n, p for expressions. The syntax definition

$$M ::= K \mid \textbf{let } x = M \textbf{ in } N \mid \textbf{lambda } x.M$$

introduces the new function symbols $K: Expr$, $\textbf{let}: Id \times Expr \times Expr \to Expr$ and $\textbf{lambda}: Id \times Expr \to Expr$.

3 Example semantics

3.1 A basic functional language

In this section we will give the semantics for a simple functional language as a signature and a set of rules of an evaluation system.

Signature We use the signature with sorts *Id* and *Expr*. We use the following symbols (possibly primed): x, y, z for identifiers; M, N, m, n, p, f, k for expressions and

$$M ::= 0 \mid succ \mid plus \mid Y \mid \ldots \mid MN \mid \textbf{let } x = M \textbf{ in } N \mid \textbf{lambda } x.M \mid m.n$$

To be explicit, MN means $apply(M, N)$. We explain $m.n$ below.

We need a new unary predicate over expressions *closed* M— (informally) M has no free variables, except ones assumed to be closed. The evaluation of a **let** and the application of a **lambda** expression (assuming call by value) are formulated in terms of our substitution operation.

Since our language allows a lambda expression to appear as the result of evaluating an expression, we must ensure that it carries with it enough information about the values of its free variables to enable it to be applied in any context, so we use Landin's notion of "closure" (Landin [7]). Our Natural Deduction technique really corresponds to stack discipline and we cannot expect it to handle function values without some further device. We do not have an explicit environment, instead we give rules which successively bind the variables appearing in the body until the body is "closed". We have to define closed, essentially by induction on the syntax. This means that for each syntactic constructor (let, lambda and so on) we need not only a rule for $\Rightarrow$ but also a rule for closed. The predicate closed conveys the binding nature of the operators; it really belongs to the static semantics.

We have to give rules for the application of closures and primitive functions to argument values. For this we introduce a new syntactic construct "$m.n$" for such applications with appropriate rules. These rules allow us to take a closure to pieces and apply its body in the correct context .

An illustrative proof follows the rules.

Evaluation rules

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n \quad m.n \Rightarrow p}{MN \Rightarrow p}$$

$$\frac{\mathit{value}\ n \quad [p/y](f.n) \Rightarrow m}{([p/y]f).n \Rightarrow m}$$

$$\frac{\mathit{value}\ n \quad [n/x]M \Rightarrow m}{(\textbf{lambda } x.M).n \Rightarrow m}$$

Example of "delta rules" for primitive functions :-

$$\frac{}{\mathit{value}\ 0}$$

$$\frac{}{\mathit{value}\ n} \qquad \frac{\mathit{value}\ m}{\mathit{succ}.n \Rightarrow \mathit{succ}.n}$$

$$\frac{}{\mathit{value}\ \mathit{plus}} \qquad \frac{\mathit{value}\ m}{\mathit{plus}.m \Rightarrow \mathit{plus}.m}$$

$$\frac{\mathit{value}\ n}{(\mathit{plus}.0).n \Rightarrow n} \qquad \frac{(\mathit{plus}.m).n \Rightarrow p}{(\mathit{plus}.(\mathit{succ}.m)).n \Rightarrow \mathit{succ}.p}$$

$$\frac{}{\mathit{value}\ Y}$$

$$\frac{\mathit{value}\ f}{Y.f \Rightarrow Y.f} \qquad \frac{(f.(Y.f)).m \Rightarrow n}{(Y.f).m \Rightarrow n}$$

Rules for evaluating lambda expressions to closures :-

$$\frac{\begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ \mathit{closed}\ M\end{array}}{\mathbf{lambda}\ x.M \Rightarrow (\mathbf{lambda}\ x.M)}$$

$$\frac{\begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ y \Rightarrow p\end{array} \qquad \mathbf{lambda}\ x.M \Rightarrow m}{\mathbf{lambda}\ x.M \Rightarrow [p/y]m}$$

The reader might like to compare these with the usual treatment using an explicit environment, ρ. For example

$$\frac{\rho \vdash N \Rightarrow n \quad \rho[n/x] \vdash M \Rightarrow m}{\rho \vdash \mathbf{let}\ x = N\ \mathbf{in}\ M \Rightarrow m}$$

$$\frac{\rho \vdash M \Rightarrow m \quad \rho \vdash N \Rightarrow n \quad m.n \Rightarrow p}{\rho \vdash MN \Rightarrow p}$$

In the latter rule the environment is not used; it is simply passed down. This is implicit in our Natural Deduction formulation. We only mention the environment when it is used and therefore succeed in eliminating various redundancies.

Rules for closed

$$\frac{\mathit{closed}\ n \qquad \begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ \mathit{closed}\ M\end{array}}{\mathit{closed}([n/x]M)}$$

$$\frac{}{\mathit{closed}\ 0}$$

and similarly for the other constants.

$$\frac{\mathit{closed}\ M \quad \mathit{closed}\ N}{\mathit{closed}(MN)}$$

$$\frac{\mathit{closed}\ m \quad \mathit{closed}\ n}{\mathit{closed}(m.n)}$$

$$\frac{\mathit{closed}\ N \qquad \begin{array}{c}(\mathit{closed}\ x)\\ \vdots\\ \mathit{closed}\ M\end{array}}{\mathit{closed}(\mathbf{let}\ x = N\ \mathbf{in}\ M)}$$

$$\frac{\begin{matrix}(closed\ x)\\ \vdots \\ closed\ M\end{matrix}}{closed(\mathbf{lambda}\ x.M)}$$

Rule for value

$$\frac{M \Rightarrow m}{value\ m}$$

Example of evaluation As an example of proofs using these rules we evaluate ($\mathbf{let}\ y = 2\ \mathbf{in}\ \mathbf{lambda}\ x.x+y$)1. We use two Lemmas, A and B, to simplify the layout. They are proved below. We will omit the proofs of such exciting facts as "*value* 1" and "$1 + 2 \Rightarrow 3$".

Main evaluation proof We use $\boldsymbol{\lambda}$ for **lambda** to save space.

$$\frac{2 \Rightarrow 2 \quad \dfrac{value\ 2 \quad \dfrac{\dfrac{[y' \Rightarrow 2]_{(1)}}{A}}{\boldsymbol{\lambda} x.x + y' \Rightarrow [2/y'](\boldsymbol{\lambda} x.x + y')}}{[2/y](\boldsymbol{\lambda} x.x + y) \Rightarrow [2/y](\boldsymbol{\lambda} x.x + y)}\,(1)}{\mathbf{let}\ y = 2\ \mathbf{in}\ \boldsymbol{\lambda} x.x + y \Rightarrow [2/y](\boldsymbol{\lambda} x.x + y)} \quad 1 \Rightarrow 1 \quad \dfrac{B}{([2/y]\boldsymbol{\lambda} x.x + y).1 \Rightarrow 3}$$
$$\overline{\qquad (\mathbf{let}\ y = 2\ \mathbf{in}\ \boldsymbol{\lambda} x.x + y)1 \Rightarrow 3 \qquad}$$

Lemma A Show that if $y' \Rightarrow 2$ then $(\boldsymbol{\lambda} x.x + y') \Rightarrow [2/y'](\boldsymbol{\lambda} x.x + y')$.

$$\frac{y' \Rightarrow 2 \quad \dfrac{\dfrac{[closed\ (y')]_{(4)} \quad [closed(x)]_{(5)}}{closed\ (x + y')}}{\boldsymbol{\lambda} x.x + y' \Rightarrow \boldsymbol{\lambda} x.x + y'}\,(5)}{\boldsymbol{\lambda} x.x + y' \Rightarrow [2/y'](\boldsymbol{\lambda} x.x + y')}\,(4)$$

Lemma B Show $([2/y]\boldsymbol{\lambda} x.x + y).1 \Rightarrow 3$.

$$\frac{value\ 1 \quad \dfrac{value\ 2 \quad \dfrac{value\ 1 \quad \dfrac{[x' \Rightarrow 1]_{(1)} \quad [y' \Rightarrow 2]_{(1)}}{x' + y'}}{(\boldsymbol{\lambda} x.x + y').1 \Rightarrow 3}}{[2/y]((\boldsymbol{\lambda} x.x + y).1) \Rightarrow 3}}{([2/y]\boldsymbol{\lambda} x.x + y).1 \Rightarrow 3}$$

3.2 Complex Declarations

This is the first example which necessitates of multisorted alpha notation and multisorted evaluation systems. We now consider allowing **let** to be followed by a complex declaration, formed with "**and**" (parallel declarations) and ";" (sequential declarations). Such declarations will form a new syntax class and will need to have as values just the environments which we have otherwise succeeded in eliminating. This seems unavoidable for languages, such as Standard ML, which permit such a declaration feature. However these "environments" only appear as values (declaration values) and then consist only of the sublist of variables which are mentioned in the declaration.

Signature We extend the previous signature as follows. We introduce a new *Expr*-like sort *Declaration* with a new evaluation predicate $\Rightarrow_d$, a new closure predicate $closed_d$ and a new substitution function $[\ /\ ]_d$: $Expr \times Id \times Declarations \to Declarations$. Since we use multisorted alpha notation we also have the functions α_d,$\underline{\alpha}_d$: $Id \times Id \times Declarations \to Declarations$. We use the following symbols: R, S, r, s, t for declarations;

$$R ::= x = M \mid R \textbf{ and } S \mid R; S \mid \{r\}S \mid r@s$$

We generalise the syntax class *Expr*, introducing a new **let** and $\{\ \ \}$:

$$M ::= \textbf{let } R \textbf{ in } N \mid \{r\}M$$

Finally we introduce the new rules for $\Rightarrow$,$\Rightarrow_d$,*closed* and $closed_d$. It is worth while remarking that by having introduced declarations as an *Expr*-like sort we do not have to explain how to evaluate expressions of the form $[n/x]S \Rightarrow_d s$, since the rule for this new sort of substitution follows from our earlier conventions. We will give it anyway, although it is just a polymorphic variant of the one we discussed for expressions, for the benefit of the reader.

We want a rule for substituting expressions for identifiers in declarations. The rule is

$$\frac{\textit{value } n \quad \begin{matrix}(x' \Rightarrow n) \\ \vdots \\ \alpha_{x'/x}S \Rightarrow_d \underline{\alpha}_{x'/x}s\end{matrix}}{[n/x]_d S \Rightarrow_d s} \quad x' \text{ is a new variable}$$

provided that x' is a new variable, that is it does not occur in any assumption except the ones on the top line nor in s, x, S, N and n. We will omit subscripts to predicates in the two following lists of rules.

Evaluation rules

$$\frac{M \Rightarrow m}{x = M \Rightarrow m/x}$$

$$\frac{R \Rightarrow r \quad S \Rightarrow s}{R \textbf{ and } S \Rightarrow r@s}$$

$$\frac{R \Rightarrow r \quad \{r\}S \Rightarrow s}{R; S \Rightarrow r@s}$$

$$\frac{[n/x]S \Rightarrow s}{\{n/x\}S \Rightarrow s}$$

$$\frac{\{r\}(\{t\}S) \Rightarrow s}{\{r@t\}S \Rightarrow s}$$

$$\frac{[n/x]M \Rightarrow m}{\{n/x\}M \Rightarrow m}$$

$$\frac{R \Rightarrow r \quad \{r\}M \Rightarrow m}{\textbf{let } R \textbf{ in } M \Rightarrow m}$$

$$\frac{\{r\}(\{s\}M) \Rightarrow m}{\{r@s\}M \Rightarrow m}$$

Rules for closed (This is just a selection)

$$\frac{\mathit{closed}([n/x]M)}{\mathit{closed}(\{n/x\}M)}$$

$$\frac{R \Rightarrow r \quad \mathit{closed}(\{r\}M)}{\mathit{closed}(\textbf{let } R \textbf{ in } M)}$$

$$\frac{\mathit{closed}(\{r\}(\{s\}M))}{\mathit{closed}(\{r \textbf{ and } s\}M)}$$

3.3 An assignment language

We will now discuss the operational semantics of a language with imperative features. We will give two alternative signatures, both extending the one defined in the previous section, with a new kind of expressions $[C]M$, the intended meaning of this being "evaluate M after doing the command C". In the first signature we will introduce a new *Expr*-like sort *Command*. These commands do not have a value of their own and their effect appears only in relation to an expression. The second signature is more denotational in nature. It is suggested by the natural remark that there is no conceptual difference between a command and a complex declaration, both denote transformations of an environment. In this case we merely extend the class of declarations. In this case commands have an explicit value. We will only sketch this signature.

A first signature We extend the previous signature as follows. We introduce a new *Expr*-like sort *Command*. We do not need to introduce a new substitution function right now. We use the following symbols: C, D for commands;

$$C ::= x := M \mid C\,; D \mid \textbf{if } M \textbf{ do } C \mid \textbf{while } M \textbf{ do } C$$

We extend the syntax class M to be $M ::= [C]M$. As remarked earlier, the intended meaning of this new kind of expression is "evaluate M after doing the command C". For example if $x \neq y$ then $[x := 1; y := 2]x + y$ evaluates to 3. These expressions have no side effect. The use of [ ] for commands should not be confused with the notation for substitution. However there is a suggestive analogy. We introduce a new predicate over *Command*, $\mathit{closed}_{\mathit{comm}}(C)$, which we will write simply as $\mathit{closed}(C)$. Finally we introduce the new rules for $\Rightarrow$ and *closed*. As usual we omit subscripts.

Evaluation rules

$$\frac{M \Rightarrow m \quad [m/x]N \Rightarrow n}{[x := M]N \Rightarrow n}$$

$$\frac{[C]([D]M) \Rightarrow m}{[C;D]M \Rightarrow m}$$

$$\frac{N \Rightarrow \mathit{true} \quad [C]M \Rightarrow m}{[\mathbf{if}\ N\ \mathbf{do}\ C]M \Rightarrow m}$$

$$\frac{N \Rightarrow \mathit{false} \quad M \Rightarrow m}{[\mathbf{if}\ N\ \mathbf{do}\ C]M \Rightarrow m}$$

$$\frac{[\mathbf{if}\ N\ \mathbf{do}\ (C;\mathbf{while}\ N\ \mathbf{do}\ C)]M \Rightarrow m}{[\mathbf{while}\ N\ \mathbf{do}\ C]M \Rightarrow m}$$

Rules for closed

$$\frac{\mathit{closed}\ C \quad \mathit{closed}\ M}{\mathit{closed}\ [C]M}$$

$$\frac{\mathit{closed}\ M}{\mathit{closed}(x := M)}$$

$$\frac{\mathit{closed}\ C \quad \mathit{closed}\ D}{\mathit{closed}(C;D)}$$

$$\frac{\mathit{closed}\ N \quad \mathit{closed}\ C}{\mathit{closed}(\mathbf{if}\ N\ \mathbf{do}\ C)}$$

$$\frac{\mathit{closed}\ N \quad \mathit{closed}\ C}{\mathit{closed}(\mathbf{while}\ N\ \mathbf{do}\ C)}$$

An alternative signature We do not extend the signature with a new *Expr*-like sort *Command* as we did before. We extend instead the *Expr*-like sort *Declarations*. We use the following symbols: S, R, C, D for complex declarations; and

$$S ::= x := M \mid C\,;D \mid \mathbf{if}\ M\ \mathbf{do}\ C \mid \mathbf{while}\ M\ \mathbf{do}\ C \mid [\quad]$$

The intended meaning of $[\quad]$ is the empty declaration. We extend the syntax class M to $M ::= [C]M$.

Evaluation rules

$$\frac{C \Rightarrow s \quad \{s\}M \Rightarrow m}{[C]M \Rightarrow m}$$

$$\frac{M \Rightarrow m}{[\ \ ]M \Rightarrow m}$$

$$\frac{M \Rightarrow \mathit{true} \quad C \Rightarrow s}{\textbf{if } M \textbf{ do } C \Rightarrow s}$$

$$\frac{M \Rightarrow \mathit{false}}{\textbf{if } M \textbf{ do } C \Rightarrow [\ \ ]}$$

$$\frac{\textbf{if } N \textbf{ do } (C; \textbf{while } N \textbf{ do } C) \Rightarrow s}{\textbf{while } N \textbf{ do } C \Rightarrow s}$$

3.4 Expressions with side effects

How can we extend the system to deal with expressions which may have side effects? Here are some tentative thoughts. In the above C had a side effect but with the given semantics for the functional language $[C]M$ had no side effects. Now let us change the semantics of the functional language to allow expressions, M, to have side effects. To accomplish this we adopt the following device: write $[M]N$ to mean the value of N after evaluating the expression M. Now for example $[M+N]P$ has the same value as $[M]([N]P)$ and $M+N \Rightarrow p$ if $M \Rightarrow m$ and $[M]N \Rightarrow n$ and $m+n \Rightarrow p$.

The revised semantic rules might be as follows. We do not give them all, just enough to illustrate the idea.

$$\frac{N \Rightarrow n}{[x]N \Rightarrow n}$$

$$\frac{N \Rightarrow n}{[0]N \Rightarrow n}$$

and similarly for the other constants.

$$\frac{N \Rightarrow n}{[\textbf{lambda } x.M]N \Rightarrow n}$$

The rule for evaluating an application becomes

$$\frac{M \Rightarrow m \quad [M]N \Rightarrow n \quad m.n \Rightarrow p}{MN \Rightarrow p}$$

$$\frac{P \Rightarrow p}{[m.n]P \Rightarrow p} \qquad (m.n \text{ has no side effect})$$

$$\frac{[C]([M]N) \Rightarrow n}{[[C]M]N \Rightarrow n}$$

3.5 Exceptions

To handle exceptions we introduce a new basic syntax class e— exception names and a new ternary judgement, "$\_ \Rightarrow \_!\_$". $M \Rightarrow e!m$ — evaluating M produces an exception e with value m. The exception is raised by "raise" and the value m is caught by a "handle" clause (we follow the notation of Standard ML). The extension to the syntax is

$$M ::= \textbf{raise}\ e\ M \mid M\ \textbf{handle}\ ex\ \textbf{in}\ N$$

The handle expression is the same as M if M evaluates normally or evaluates to an exception other than e; but if M evaluates to exception e with value m then the result is found by evaluating N with x bound to m.

Example:[4]

$$\textbf{raise}\ \ help\ 999 \Rightarrow\ \ help\ !999$$

$$1\ \textbf{handle}\ \ help\ x\ \textbf{in}\ x \times 10 \Rightarrow 1$$

$$(1 + (\textbf{raise}\ \ help\ 999))\ \textbf{handle}\ \ help\ x\ \textbf{in}\ x \times 10 \Rightarrow 9990$$

We need rules for **raise** and **handle**. We must also add rules to cope with the exception cases of application, **let** and other constructions above; we just give the rules for application as the others follow a similar pattern. We get a lot more rules, but the normal (non-exception) rules are unchanged. This makes it easier to understand the semantics; you can study the normal rules first. We do not discuss exception declarations but only global exceptions: **handle** does not bind exception names. Subscripts are omitted.

$$\frac{M \Rightarrow m}{\textbf{raise}\ e\ M \Rightarrow e!m}$$

$$\frac{M \Rightarrow e!m}{MN \Rightarrow e!m}$$

$$\frac{M \Rightarrow e!m \quad N \Rightarrow n}{MN \Rightarrow n}$$

$$\frac{M \Rightarrow m \quad N \Rightarrow e!n}{MN \Rightarrow e!n}$$

$$\frac{M \Rightarrow m}{M\ \textbf{handle}\ ex\ \textbf{in}\ N \Rightarrow m}$$

$$\frac{M \Rightarrow e!m \quad [m/x]N \Rightarrow n}{\textbf{handle}\ ex\ \textbf{in}\ N \Rightarrow n}$$

[4]The italian reader can get an example with the same connotation by replacing the string 999 with 113 throughout the example

$$\frac{M \Rightarrow e!m \quad [m/x]N \Rightarrow e'!n}{M \textbf{ handle } ex \textbf{ in } N \Rightarrow e'!n}$$

$$\frac{M \Rightarrow e!m}{M \textbf{ handle } e'x \textbf{ in } N \Rightarrow e!m}$$

The second and third rules for apply are very similar; since we have different judgements for normal and exceptional evaluation, we cannot combine them. The same is true for **handle**. The use of subsorts in the meta language would enable a more economical treatment here; we would then have subsorts of normal and exceptional values instead of two judgements, compare [4]. We lose the advantage that normal values can be treated without mentioning exceptions. The Standard ML semantics [10] is slightly informally presented but seems to implicitly use subsorts.

3.6 Recursive function definitions

Amongst the primitive operations defined above was Y, the fixed point operator. This gives us the ability to define recursive functions by applying Y to a functional. However we can define recursive definitions more directly by a "**letrec**" construction. This has an unwinding rule rather like the one for **while** , but a little more subtle. We will treat recursive functions here, rather than recursive procedures. The new syntax, using f for function identifiers, is:

$$M ::= \textbf{letrec} f(x) = M \textbf{ in } N$$

The rules are

$$\frac{\textbf{let } f = (\textbf{lambda } x.\textbf{letrec } f(x) = N \textbf{ in } N) \textbf{ in } M \Rightarrow p}{\textbf{letrec } f(x) = N \textbf{ in } M \Rightarrow p}$$

$$\frac{\begin{matrix}(\mathit{closed}\ f \quad \mathit{closed}\ x) & (\mathit{closed}\ f) \\ \vdots & \vdots \\ \mathit{closed}\ N & \mathit{closed}\ M\end{matrix}}{\mathit{closed}(\textbf{letrec } f(x) = N \textbf{ in } M)}$$

Mysterious? Let us try an example. We will do it upside down.

$\textbf{letrec } f(x) = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1) \textbf{ in } f(1) \Rightarrow ?$

$\textbf{let } f = \textbf{lambda } x.(\textbf{letrec } f(x) = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1) \textbf{ in}$
$(\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1) \textbf{ in } f(1) \Rightarrow ?$

$[1/x]\textbf{letrec } f(x) = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1) \textbf{ in}$
$(\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1)(x) \Rightarrow ?$

$\textbf{letrec } f(x) = (\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1) \textbf{ in}$
$(\textbf{if } x = 0 \textbf{ then } 0 \textbf{ else } f(x-1)+1)(1) \Rightarrow ?$

3.7 Local declarations and procedures with parameters

In this example we illustrate the semantics of a language with local variables in commands and with procedure declaration facilities. More precisely we discuss procedures with one parameter, passed by value, and possibly with local variables. We do not consider kinds of parameters other than value ones or procedures as parameters. We do not address the issue of recursive procedures here. Also this example uses multisorted alpha notation. It is perhaps the most involved example so far. We need to introduce the new substitution function corresponding to the syntax class *Command* and hence the appropriate rule to manipulate it.

Signature We extend the signature of the assignment language as follows. We introduce a new substitution function, which enables us to take care of local variables, $[\ \ /\ \ ]_{Comm}$: *Expr* $\times$ *Id* $\times$ *Command* $\to$ *Expr*. Correspondingly we add the appropriate substitution rule (dropping subscripts).

$$\frac{\begin{array}{c} \\ \\ \mathit{value}\ n \end{array} \quad \begin{array}{c}(x' \Rightarrow m)\\ \vdots \\ [\alpha_{x'/x}C]N \Rightarrow \underline{\alpha}_{x'/x}n\end{array}}{[[m/x]C]N \Rightarrow n} \quad x' \text{ is a new variable}$$

provided that x' is a new variable, that is it does not occur in any assumption except the ones on the top line nor in m, x, C, N and n. Of course we have in the background also the function α_d: *Id* $\times$ *Id* $\times$ *Command* $\to$ *Command* .

We introduce a new *Expr*-like sort *Procedures* and an *Id*-like sort *Procedure_names*, with a corresponding evaluation predicate $\Longrightarrow_{proc}$. We use the following symbols: Q, h for *Procedures*; P for *Procedure_names*.

$$P ::= P_0 \mid \ldots \mid P_k$$

$$Q ::= \textbf{lambda}\ x.C$$

We generalise the syntax class *Command* to

$$C ::= \textbf{begin}\ \ \textbf{new}\ x = M\ \textbf{in}\ D\ \textbf{end}\ \mid \textbf{proc}\ P(x) = C\ \textbf{in}\ D \mid P(M)$$

In giving the rules we will omit subscripts.

The mechanism for procedure call is given by $[P/h]_{Proc}C$, where we use a new substitution function $[\ \ /\ \ ]_{Proc}$: *Proc* $\times$ *Id* $\times$ *Command* $\to$ *Expr* for substituting procedures for procedure identifiers in commands. We omit the rule to which it obeys since it is similar to the one for $[m/x]C$. We do not need to introduce procedure closures. Finally we introduce the new rules for $\Rightarrow$ and *closed* . As usual we drop subscripts.

Evaluation rules

$$\frac{M \Rightarrow m \quad [[m/x]C]N \Rightarrow n}{[\textbf{begin}\ \ \textbf{new}\ x = M; C\ \textbf{end}\ ]N \Rightarrow n}$$

$$\frac{[[\mathbf{lambda}\ x.C/P]D]N \Rightarrow n}{[\mathbf{proc}\ P(x) = C\ \mathbf{in}\ D]N \Rightarrow n}$$

$$\frac{P \Longrightarrow \mathbf{lambda}\ x.C \quad M \Rightarrow m, [[m/x]C]N \Rightarrow n}{[P(M)]N \Rightarrow n}$$

Rules for closed

$$\frac{closed\ n \quad \begin{array}{c} closed\ x \\ \vdots \\ closed\ C \end{array}}{closed([n/x]C)}$$

$$\frac{closed\ M \quad \begin{array}{c} closed\ x \\ \vdots \\ closed\ C \end{array}}{closed(\mathbf{begin}\ \ \mathbf{new}\ x = M; C\ \mathbf{end}\ )}$$

$$\frac{closed\ [[\mathbf{lambda}\ x.C/P]D]}{closed\ [\mathbf{proc}\ P(x) = C\ \mathbf{in}\ D]}$$

$$\frac{closed\ M \quad closed\ N}{closed\ [P(M)]N}$$

$$\frac{\begin{array}{c} closed(x) \\ \vdots \\ closed(C) \end{array} \quad closed([D]N)}{closed([[\mathbf{lambda}\ x.C/P]D]N)}$$

An alternative Signature Again we can explain local variables in commands and hence procedures by reducing them to complex declarations. First we introduce procedures and procedure names as before. Then we extend the syntax class of complex declarations as follows:

$$S ::= \mathbf{begin}\ \ \mathbf{new}\ x = M\ \mathbf{in}\ R\ \mathbf{end}\ \ |\ \mathbf{proc}\ P(x) = R\ \mathbf{in}\ S$$

and add the rules

$$\frac{M \Rightarrow m \quad m/xS \Rightarrow s}{\mathbf{begin}\ \ \mathbf{new}\ x = M; S\ \mathbf{end}\ \Rightarrow s}$$

$$\frac{[\mathbf{lambda}\ x.S/P]D \Rightarrow n}{\mathbf{proc}\ P(x) = S\ \mathbf{in}\ D \Rightarrow n}$$

$$\frac{P \Longrightarrow \mathbf{lambda}\ x.S \quad M \Rightarrow m \quad m/xS \Rightarrow s}{P(M) \Rightarrow s}$$

In this case we have just to introduce the substitution rule for procedures. Again it is a polymorphic variant of the one for declarations

4 Definition of Evaluation Systems and Alpha Notation in the Edinburgh Logical Framework

In this section we outline two alternative definitions of an Evaluation System with a minimal signature in the Edinburgh Logical Framework. This is a signature with two sorts Id and $Expr$, three function symbols α, $\underline{\alpha}$: $Id \to Id \to Expr \to Expr$, and $[\ \ /\ \ ]$: $Expr \to Id \to Expr \to Expr$ and two predicates $value$ over $Expr$ and $\Rightarrow$ over $Expr \times Expr$. The first encoding introduces explicit formal counterparts for the operators α, $\underline{\alpha}$ of alpha notation and therefore counts also as a rigorous explanation of such a notation. The second encoding does not have formal counterparts for α, $\underline{\alpha}$. Hence it illustrates how evaluation systems can be used without the need of alpha notation if one is willing to work with the full higher order power of ELF.

The first ELF signature for the minimal Evaluation System In order to encode in ELF such an Evaluation System we proceed as follows. First of all we introduce an ELF type corresponding to the collection of sorts and a type constructor, *Term*, defined on sorts.

$$Sorts : \ Type$$

$$Term : \ Sorts \ \to \ Type$$

Then we introduce constants corresponding to Id and $Expr$ and two new constants. The first $\supset$, is intended to denote the higher order sort constructor (written as an infix and bracketed to the right), while the second is intended to denote syntactic application.

$$Id : Sorts$$

$$Exp\ r\colon Sorts$$

$$\supset\colon Sorts \ \to Sorts \ \to Sorts$$

$$app\colon \Pi_{s,t}\colon Sorts. Term\ (s \supset t) \to Term\ (s) \to Term\ (t)$$

Corresponding to a function in the signature we declare a constant of type $Term(s)$ for the appropriate s, that is

$$\alpha, \underline{\alpha}\colon Term\ (Id \supset Id \supset Expr \supset Expr)$$

$$expr : Term\ (Id \supset Expr)$$

$$[\ \ /\ \ ]\colon Term\ (Expr \supset Id \supset Expr \supset Expr)$$

Corresponding to the class of formulae in the system we introduce an ELF type $Form : \ Type$. Corresponding to a predicate in the signature we introduce a functional constant with domain $Term(s)$ for appropriate sorts s, that is $value : Term\ (Expr) \ \to \ Form$ and $\Rightarrow\colon Term\ (Expr) \to Term\ (Expr) \to Form.$

Finally we introduce a judgement forming operator, asserting the truth of formulae in the evaluation system

$True : Form \;\rightarrow\; Type$.

We now encode the rules for the replacement operators α and $\underline{\alpha}$. This is the most elaborate part. We have to introduce in the ELF signature a number of new predicates, new constants, new formula forming operators, and new rules governing the provability of these new formulae. More precisely we define

$\in : \Pi_s : Sorts.\,Term(Id) \rightarrow Term(s) \rightarrow \;Type$

$\notin : \Pi_s : Sorts.\,Term(Id) \;\rightarrow Term(s) \rightarrow \;Type$

The judgement $x \in M$ says that the identifier x occurs in M, while $x \notin M$ says that it does not occur. To translate the rules of Alpha notation we introduce constants of the types which follow. (For ease of reading we will write rules using a somewhat more suggestive notation than the ELF Π and $\rightarrow$. We will use the notation τs instead of $Term(s)$ and subscript arguments which are sorts to other functions).

$$\frac{x \in_s M}{x \in_t app_{st} N M} \quad s,t : Sorts \quad x : \tau Id \;\; M : \tau s \;\; N : \tau s \supset t$$

$$\frac{x \in_s \supset tN}{x \in_t app_{st} N M} \quad s,t : Sorts \quad x : \tau Id \;\; M : \tau s \;\; N : \tau s \supset t$$

$$\frac{x \notin_s M \quad x \notin_{s \supset t} N}{x \notin_t app_{st} N M} \quad s,t : Sorts \quad x : \tau Id \;\; M : \tau s \;\; N : \tau s \supset t$$

$$\frac{}{x \in_{Id} x} \quad x : \tau Id$$

$$\frac{y \notin_{Id} x}{x \notin_{Id} y} \quad x,y : \tau Id$$

$$\frac{}{x \notin_{Id \supset Expr \supset Expr \supset Expr} [\;\;/\;\;]} \quad x : \tau Id$$

$$\frac{}{x \notin_{Id \supset Expr}\, expr} \quad x : \tau Id$$

$$\frac{}{x \notin_{Id \supset Id \supset Expr \supset Expr}\, \alpha} \quad x : \tau Id \text{ (and similarly for } \underline{\alpha}\text{)}$$

When we come to add identifier constants to the signature, for each pair of distinct constants, i,j in $Term(Id)$, we will need a rule,

$$\frac{}{\notin_{Id} j}.$$

We are now ready to illustrate how to encode the set of rules about the α and $\underline{\alpha}$ operators in ELF. We introduce constants of the following ELF types.

$$\frac{x \notin_{Expr} F(y) \quad True(G(F(y)))}{True(G(\alpha_{y/x} F(x)))}$$

$$\frac{y \notin_{Expr} F(x) \quad True(G(F(y)))}{True(G(\underline{\alpha}_{y/x} F(x)))}$$

In both rules $F\colon \tau Id \rightarrow \tau Expr$, $G\colon \tau Expr \rightarrow Form$, and x, $y\colon \tau Id$. In the last rule we have omitted *app* before α and $\underline{\alpha}$ for ease of reading.

The translations of the first rule specific to evaluation systems in ELF is straightforward:

$$\frac{True(value\ (n))}{True(n \Rightarrow n)} \quad n\colon \tau Expr$$

The encoding of the substitution rule is instead quite a delicate matter. One has to reflect in the system the fact that the variable local to the minor premise is indeed different from any other variable in the system. This can be achieved using the higher order nature of ELF by allowing to establish the minor premise assuming appropriate rules which are then discharged by an application of the substitution rule itself. This is an interesting example of a second order rule, i.e. a rule with hypothetical rule assumptions. We omit the *Id* and *Expr* subscripts on $\in$; they are *Id* except before M or m.

$$\frac{True(value(n)) \qquad \dfrac{True(x' \Rightarrow n) \quad \dfrac{w \in x}{w \notin x'}\, w\colon \tau Id \quad \dfrac{w \in M}{w \notin x'}\, w\colon \tau Id \quad \dfrac{w \in m}{w \notin x'}\, w\colon \tau Id}{True(\alpha_{x'/x} M \Rightarrow \underline{\alpha}_{x'/x} m)}\ x'\colon \tau Id}{True([x/n]M \Rightarrow m)} \quad \begin{array}{r} x\colon \tau Id \\ M, m, n\colon \tau Id \end{array}$$

In the last rule for ease of reading we have omitted *expr*, also *app* before α, $\underline{\alpha}$ and $[\ \ /\ \ ]$. As one can see now, the main benefit of using directly alpha notation is that in the informal substitution rule the higher order syntactic notions $M, m\colon \tau Id \rightarrow Expr$ and the hypothetical rules premises do not appear.

This ELF signature can be easily extended to the case of multi-sorted Evaluation systems. Care has to be taken only in encoding hypothetical premises in the informal rules (e.g. substitution rules or rules for closed). One has to introduce hypothetical rule premises analogous to the ones in the rule above.

Another ELF signature for the minimal Evaluation System This signature is much the same as the one we defined above. We have all the constants we had in the previous signature but the ones which involve α $\underline{\alpha}$. In particular we have:

$$\begin{array}{rcl} Sorts & : & Type \\ Term & : & Sorts \rightarrow Type \\ Form & : & Type \\ True & : & Form \rightarrow Type \end{array}$$

$$
\begin{array}{rcl}
Id & : & Sorts \\
Expr & : & Sorts \\
\supset & : & Sorts \to Sorts \to Sorts \\
app & : & \Pi_{s,t}{:}\, Sorts. Term(s \supset t) \to Term(s) \to Term(t) \\
expr & : & Term(Id \supset Expr) \\
[\ \ /\ \ ] & : & Term(Expr \supset Id \supset Expr \supset Expr) \\
value & : & Term(Expr) \to Form \\
\Rightarrow & : & Term(Expr) \to Term(Expr) \to Form \\
\in & : & \Pi_s{:}\, Sorts. Term\ (Id) \to Term\ (s) \to Type \\
\notin & : & \Pi_s{:}\, Sorts. Term\ (Id) \to Term\ (s) \to Type
\end{array}
$$

$$
\frac{x \in_s M}{x \in_t app_{st} N M} \quad s, t{:}\, Sorts, x{:}\, \tau Id, M{:}\, \tau s, N{:}\, \tau s \supset t
$$

$$
\frac{x \in_{s \supset t} N}{x \in_t app_{st} N M} \quad s, t{:}\, Sorts, x{:}\, \tau Id, M{:}\, \tau s, N{:}\, \tau s \supset t
$$

$$
\frac{x \notin_s M \quad x \notin_{s \supset t} N}{x \notin_t app_{st} N M} \quad s, t{:}\, Sorts, x{:}\, \tau Id, M{:}\, \tau s, N{:}\, \tau s \supset t
$$

$$
\frac{}{x \in_{Id} x} \quad x{:}\, \tau Id
$$

$$
\frac{y \notin_{Id} x}{x \notin_{Id} y} \quad x, y{:}\, \tau Id
$$

$$
\frac{}{x \notin_{Id \supset Expr \supset Expr \supset Expr} [\ \ /\ \ ]} \quad x{:}\, \tau Id
$$

$$
\frac{}{x \notin_{Id \supset Expr}\, expr} \quad x{:}\, \tau Id
$$

$$
\frac{True(value(n))}{True(n \Rightarrow n)} \quad n{:}\, \tau Expr
$$

The encoding of the substitution rule is much more involved. In order to get the same conclusion as before we need to have more premises. We omit the *Id* and *Expr* subscripts on $\in$; they are *Id* except before M or m.

$$
\frac{True(value(n)) \quad x \notin M \quad x \notin m \quad \dfrac{\dfrac{w \in x}{w \notin x'}\ w{:}\, \tau Id\ \dfrac{w \in M}{w \notin x'}\ w{:}\, \tau Id\ \dfrac{w \in m}{w \notin x'}\ w{:}\, \tau Id}{True(M(x') \Rightarrow m(x'))}\ x'{:}\, \tau Id}{True([x/n] M(x) \Rightarrow m(x))} \quad \begin{array}{r} x{:}\, \tau Id \\ n{:}\, \tau Expr \\ M, m{:}\, \tau Id \to Expr \end{array}
$$

As one can see now the main benefit of alpha operators in the ELF encoding is to reduce the number of premises in this rule.

5 Concluding remarks

We have shown how to define semantics of a simple but non-trivial language in our Natural Deduction style. We have not treated reference variables and data types, nor have we defined the type discipline by a static semantics. These remain to be investigated. Another area for exploration would be the application of the technique to defining logics. We would also like to consider program verification and transformation in this formalism . Although our system relies on the Edinburgh Logical Framework for a formal definition, it can be applied without explicit reference to ELF, basing it on Alpha notation.

References

[1] Rod Burstall and Furio Honsell. A natural deduction treatment of operational semantics. In *Proc. 8th Conf. on Foundations of Software Technology, Pune (India)*. Springer LNCS, 1988.

[2] Rod Burstall and Furio Honsell. A natural deduction treatment of operational semantics. LFCS, Comp. Sci. Dept., Edinburgh University, UK, 1989.

[3] Reynolds J. C. Syntactic control of interference. In *Proc. 5th Annual Symp. on Principles of Prog. Langs., Tucson,*. ACM, 1978.

[4] J.A. Goguen. Order Sorted Algebra. Technical Report 14, Computer Science Dept., University of Calif. at Los Angeles, US, 1978.

[5] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Second Annual Conference on Logic in Computer Science, Cornell, USA*, 1987.

[6] Gilles Kahn. Natural semantics. Rapport de Recherche 601, INRIA, France, 1987.

[7] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6, 1964.

[8] Ian Mason. Hoare's logic in the LF. Technical Report LFCS-87-32, Comp. Science Dept. Edinburgh University, UK, 1987.

[9] Hannan J and Miller D., Enriching a Meta-Language with Higher Order Features In Proceedings of the 1988 Workshop on Meta-Programming in Logic Programming, Bristol, UK.

[10] R. Milner. Notes on relational semantics, 1987.

[11] C.-E. Ore. On Natural Deduction Style Semantics, Environments and Stores Technical report, LFCS 89-88, Comp. Science Dept. Edinburgh University, UK, 1989.

[12] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

[13] Luo Z., Pollack R. A. and Taylor P. (1989), LEGO User Manual, LFCS, Dept. of Computer Science, Edinburgh University, UK.

[14] D. Prawitz. *Natural Deduction: A Proof-Theoretic study.* Almqvist & Wiksel, Stockholm, 1965.

Deliverables: an approach to program development in the Calculus of Constructions[1]

Rod Burstall and James McKinna[2]
Laboratory for the Foundations of Computer Science
University of Edinburgh[3]

August 7, 1990

The problem: separating proofs and programs

There are two contrasting approaches to the formal development of correct programs:

Classical: We can write a program and produce a separate proof of its correctness

Synthetic: We prove the corresponding theorem and extract from it the program (its algorithmic content)

The classical approach seems unsatisfactory in that the proof is separate from the program. It is more acceptable in a formalism such as Floyd-Hoare assertions in which local correctness statements are attached to pieces of program.

The synthetic method is attractive at first sight but there is considerable difficulty in separating the algorithmic part of the proof from the correctness part. No automatic method for doing this is known. See for example Constable *et al* (1986), Nordström, Peterssen and Smith (1990) and Paulin-Mohring (1989).

Extracting programs from proofs

Classically we look for a constructive proof of a statement of the form

$\forall x.(S(x) \rightarrow \exists y.R(x, y))$,

in Constructions notation

```
{x:i} (S x -> exists [x:i] R x y).
```

But when we do a constructive proof we wind up with a function which, given an individual `x` and a proof of `S x`, produces a pair consisting of a an individual `y` and a proof that it satisfies `R x y`. In other words we cannot get the result `y` without providing the proof of `S x`. It seems that if we try to develop a substantial program this way we need to handle proofs all the time as we attempt to calculate values: the proof and the computation are inextricably mixed.

A better idea is to look for a pair (f, p) where f is a function from individuals to individuals and p is a proof that

[1]Draft

[2]The authors gratefully acknowledge the support of the EC BRA and the SERC

[3]J.C.M.B., King's Buildings, Mayfield Rd., Edinburgh EH9 3JZ, UK;
e-mail `rb@lfcs.ed.ac.uk, jhm@lfcs.ed.ac.uk`

$\forall x.(S(x) \rightarrow R(x, f(x))),$

in Constructions notation

```
{x:i} (S x -> R x (f x)).
```

Now this is not much good if we have to revert to the classical method of developing f, the program, and p, the proof, independently.

We want to take a more categorical view and build up such program-proof pairs by composition. We will call these pairs "deliverables"; they are what a Software House should deliver to its customers, a program plus a proof in a box with the specification printed on the cover. The customer can independently check the proof and then run the program. The point is that the program part can be trivially extracted from such a pair by taking the first projection and normalising. We have carried this out in Pollack's LEGO implementation of the Calculus of Constructions, see Luo, Pollack and Taylor (1989), Coquand and Huet (1988).

Later we will note that a specification given by an input property and an output property is insufficient, and show how the ideas can be extended to specify a relation between the input and the output.

The Cartesian Closed Category of Deliverables

We will define a cartesian closed category (ccc) of deliverables. That is it will have deliverables as morphisms. For an equational definition of a ccc see Lambek and Scott (1986), Section I.3; we will follow their treatment. The ccc is built up in three stages: first a category, then adding a terminal object and binary products, then adding exponentials. At each stage we will specify the equations which have to be verified. In order to satisfy these equations we will have to assume η-conversion and surjective pairing for products (non-dependent Σ); these are not in the normalisation algorithm for our LEGO implementation of Calculus of Constructions, but it is conjectured that they could be added without loss of consistency.

A category: composition and identity

Objects A type, s, together with a set over s, $S: s \rightarrow \text{Prop}$.

Morphisms Morphisms from (s, S) to (t, T), say $\text{del}_{s,t}\ S\ T$ are pairs

- $f: s \rightarrow t$
- p, a proof that for all $x: s$, $Sx \supset T(fx)$

$\text{Del}_{s,t}\ S\ T\, f$ will be used to abbreviate

$\forall x: s.Sx \supset T(fx).$

Thus $\text{Del}_{s,t}\ S\ T$ is a set over $s \rightarrow t$. Define $\text{del}_{s,t}\ S\ T$ as the dependent sum type $\Sigma f: s \rightarrow t.\text{Del}_{s,t}\ S\ T\, f$.

In general we will write types as subscripts and feel free to drop them where they can be understood from context. We will use F, G, H to denote morphisms in this category. Sometimes we will write the object (s, S) just as S, where the s may be inferred from the definition of S. In the LEGO code we are able to drop these type parameters, taking advantage of the type inference facility.

We define the composition thus

$$\frac{(s,S) \xrightarrow{(f,p)} (t,T) \quad (t,T) \xrightarrow{(g,q)} (u,U)}{(s,S) \xrightarrow{(f \circ g,\, p * q)} (u,U)} \ \mathrm{comp}_{s,t,u}\, S\ T\ U$$

where $f \circ g$ is the composition and $p*q$ is a proof of $(\mathrm{del}_{s,u}\ S\ U)(f \circ g)$. In fact $(p*q)x = q(fx)(px)$. We will not give such proofs explicitly from now on; they are easily derived in a top down manner in LEGO, and their explicit form is not particularly illuminating.

We define the identity thus

$$\frac{}{(s,S) \xrightarrow{(\mathrm{id}_s,\, p)} (s,S)} \ \mathrm{ident}_s\, S$$

where id_s is the identity function on s, and p is a (trivial) proof that it is in $\mathrm{Del}_{s,s}\ S\ S$. Thus using our convention of omitting types we may say that, given $F\colon S \to T$ and $G\colon T \to U$ we have defined $\mathrm{comp}\ F\ G : S \to U$, and we have defined $\mathrm{ident}\ S : S \to S$.

To show that this forms a category, we must check associativity and identity.

Cartesian category: terminal object and product

Our next step is to add the additional structure of a Cartesian category, a terminal object and binary products. The calculus of constructions provided by the LEGO system has binary products for types, `s#t`, but it does not have a terminal type with exactly one element, although ML for example provides such a type and it might reasonably be expected in LEGO. We hope that this minor omission may be repaired some time. There is also a difficulty with the products as follows. The pairing operation is written (x, y) and the projections $p.1$ and $p.2$; we have $(x, y).1$ reduces to x and $(x, y).2$ reduces to y, but not $(p.1, p.2)$ reduces to p. That is, pairing lacks uniqueness.

We introduce a unit type and a void element of it by declaration, rather than by definition as we would have preferred.

```
unit:Type
void:unit
Unit [x:unit] = Q x void
```

Here "`Q`" means the Leibniz equality, defined by `Q x y` iff any property holding for `x` holds for `y`. We define the unique morphism to the terminal object thus

$$\frac{}{(s,S) \xrightarrow{(u,p)} (\mathrm{unit}, \mathrm{Unit})} \ \mathrm{term}_s\, S$$

where $u\ x = \text{void}$, and p is the (trivial) proof that $\text{Unit}(u\ x)$. We need to show that if $F\colon (s,S) \to (\text{unit}, \text{Unit})$ then $f = (u,p)$. This is only true if we add the extensionality axiom: $f = g$ if for all x, $f\ x = g\ x$, where $=$ refers to Leibniz equality.

We define the product of two objects (s,S) and (t,T) as $(s\sharp t, U)$ where $U\ z = S(z.1) \wedge T(z.2)$. We write $\text{prod}_{s,t}\ S\ T$. The projections are

$$\frac{}{\text{prod}_{s,t}\ S\ T \xrightarrow{(\pi_1,p_1)} S}\ \Pi^1_{s,t}\,S\,T \qquad \frac{}{\text{prod}_{s,t}\ S\ T \xrightarrow{(\pi_2,p_2)} T}\ \Pi^2_{s,t}\,S\,T$$

where $\pi_1\ z = z.1$ and p_1 proves that Del $(\text{prod}\ S\ T)\ S\ \pi_1$. Similarly for π_2.

We define pairing on morphisms

$$\frac{(s,S) \xrightarrow{(f,p)} (t,T) \quad (s,S) \xrightarrow{(g,q)} (u,U)}{(s,S) \xrightarrow{(\langle f,g\rangle, r)} \text{prod}_{t,u}\ T\ U}\ \text{pair}_{s,t,u} S\ T\ U$$

where $\langle f,g\rangle\ x = (f\ x, g\ x)$ and r is the associated proof.

We need to show

(a1) $\pi_1(\text{pair}\ S\ T) = S$.

(a2) $\pi_2(\text{pair}\ S\ T) = T$.

(b) $\text{pair}\ (\pi_1 S, \pi_2 S) = S$.

Equations (a1) and (a2) are easy but (b) does not hold for Leibniz equality as we have remarked.

Cartesian closed category: exponentials

To form a cartesian closed category we need to add exponential objects, with two operations apply and curry. (These are respectively the co-unit and the bijection if we define the ccc by an adjunction). The exponential of (t,T) with respect to (s,S), written $\exp_{s,t}\ S\ T$ is $(s \to t, \text{del}_{s,t}\ S\ T)$. We have

$$\frac{}{\text{prod}_{t\to s,t}(\exp_{t,s} T\ S)\ (T) \xrightarrow{(f,p)} (s,S)}\ \text{apply}_{s,t}\,S\ T$$

where $fz = (z.1)(z.2)$ and p is the associated proof.

$$\frac{\text{prod}_{u,t}\ U\ T \xrightarrow{(f,p)} (s,S)}{(u,U) \xrightarrow{(g,q)} \exp_{t,s} T\ S}\ \text{curry}_{s,t,u}\ S\ T\ U$$

where $g\ x\ y = f(x,y)$ and q is the associated proof.

These must satisfy

$$\text{apply}(\text{pair}(\text{comp}\ \pi_1(\text{curry}\ F))\ \pi_2) = F \quad \text{for all } F\colon \text{prod}\ S\ T \to U$$

$$\text{curry}(\text{apply}(\text{pair}(\text{comp}\ \pi_1\ G)\ \pi_2)) = G \quad \text{for all } G{:}\,U \to \exp\ T\ S$$

Since all the necessary equations fail to hold for `Q` the Leibniz equality we should really introduce another equality `Q'` for which they are postulated. However we have not done this in the LEGO code.

Natural number object

We now add a natural number object (indeed we will have to add an object for each inductively defined data type). This object comes with a family of morphisms embodying the induction principle for natural numbers.

```
nat:Type
zero:nat
succ:nat -> nat
Nat:nat -> Prop
Nat [n:nat] = true
```

As usual, we write "0" for "zero", and "+1" (postfix) for "succ". The natural number object is (nat, Nat).

We can define a function natrec: $t \to (t \to t) \to (\text{nat} \to t)$ polymorphic for any type t with axioms

$$\text{natrec}\ z\ s\ 0 = z$$

$$\text{natrec}\ z\ s\ (n+1) = s(\text{natrec}\ z\ s\ n)$$

$$\frac{(\text{unit}, \text{Unit}) \xrightarrow{(z,p)} (t,T) \quad (t,T) \xrightarrow{(s,q)} (t,T)}{(\text{nat}, \text{Nat}) \xrightarrow{(\text{natrec}\ z\ s\ r)} (t,T)}\ \text{Natrec}_t\ T$$

where r is the appropriate proof. In fact r uses the induction principle for nat to show that if z takes you into t and s preserves t then for any n, natrec $z\ s\ n$ is in T.

Example: the function double

We need an example to see all this at work. We will define properties Odd and Even over the natural numbers and a function double; we then show that, for any natural n, double n is even.

```
Even:nat -> Prop
Odd [n:nat] = not (Even n)
```

These will have the axioms (for all n)

$$\text{Even}\ 0$$

$$\text{Even}\ n \supset \text{Odd}\ (n+1)$$

Odd $n \supset$ Even $(n+1)$

with proofs n_0, n_1 and n_2 respectively. We define the deliverables

$$\frac{}{(\mathrm{unit}, \mathrm{Unit}) \xrightarrow{(\lambda u{:}\, \mathrm{unit}.0,\, n_0)} (\mathrm{nat}, \mathrm{Even})}\ \mathrm{zE}$$

$$\frac{}{(\mathrm{nat}, \mathrm{Even}) \xrightarrow{(\lambda n{:}\, \mathrm{nat}.n+1,\, n_1)} (\mathrm{nat}, \mathrm{Odd})}\ \mathrm{sEO}$$

$$\frac{}{(\mathrm{nat}, \mathrm{Odd}) \xrightarrow{(\lambda n{:}\, \mathrm{nat}.n+1,\, n_2)} (\mathrm{nat}, \mathrm{Even})}\ \mathrm{sOE}$$

In terms of these we define a deliverable for successor of successor

$$\mathrm{ssEE} = \mathrm{comp\ sEO\ \ sOE}$$

and a deliverable for double,

$$\mathrm{doubleNE} = \mathrm{Natrec\ Even\ zE\ ssEE}$$

. Now the typechecker can determine that doubleNE: $(\mathrm{nat}, \mathrm{Even}) \rightarrow (\mathrm{nat}, \mathrm{Even})$ as required.

Top down development

In LEGO one normally develops proofs by refinement, starting from the goal, selecting an inference rule to refine by and so producing subgoals. Although we have used the bottom-up approach in the last section, it is also possible to use the top-down refinement approach. We start with the goal

$(\mathrm{nat}, \mathrm{Even}) \rightarrow (\mathrm{nat}, \mathrm{Even})$

Then refine by the Natrec rule with the invariant $(\mathrm{nat}, \mathrm{Even})$ as parameter, getting as subgoals

$(\mathrm{unit}, \mathrm{Unit}) \rightarrow (\mathrm{nat}, \mathrm{Even})$

$(\mathrm{nat}, \mathrm{Even}) \rightarrow (\mathrm{nat}, \mathrm{Even})$

The first of these is solved by refining by zE. The second is refined by comp Even Odd Even giving as subgoals

$(\mathrm{nat}, \mathrm{Even}) \rightarrow (\mathrm{nat}, \mathrm{Odd})$

$(\mathrm{nat}, \mathrm{Odd}) \rightarrow (\mathrm{nat}, \mathrm{Even})$

These are solved by sEO and sOE respectively. Instead of refining by a deliverable we could just refine by an incomplete deliverable, leaving the proof part out but giving the function part. The proof part would appear as a pending subgoal. The LEGO system is quite flexible in the way it allows one to develop a proof, and this flexibility is also available for program development. It seems much better than being restricted to a rigid sequence of decisions, whether top down or bottom up.

Extracting the program

Extracting a program from the deliverable, for example from doubleNE, is quite easy. We just take the first component, doubleNE. But if we just print this it will print as "doubleNE.1", not very illuminating; so we have to normalise it, which is done in LEGO with the command "Normal". Indeed this is the payoff from the deliverable idea: because of the way in which we have constructed the (program, proof) pair, there is no work to do at this stage except invoking the standard normalisation algorithm.

Functional versus combinatorial notation

The expression for the deliverable doubleNE does not look like a functional program, in fact it is a combinatory expression using categorical combinators. It resembles somewhat a program in Backus's language FP, and it is not very readable. It is possible, however, to write expressions for deliverables which are very close in form to functional programs. To do this we recall the Yoneda embedding. This is a way of representing a morphism in an arbitrary category as a family of morphisms in **Set**, in fact as a polymorphic function.

Suppose $f: B \to C$ in a category $\mathbf{C}$, and let A be an object of $\mathbf{C}$. Now $\mathcal{C}[\_, B]$ and $\mathcal{C}[\_, C]$ are functors from $\mathbf{C}$ to **Set**. Then f corresponds to a natural transformation f^* from $\mathcal{C}[\_, B]$ to $\mathcal{C}[\_, C]$, defined for $g: A \to B$ by $f^* g = g \circ f$. But it has been known for some years that polymorphic functions are natural transformations. Also given such a natural transformation ϕ, we can recover a morphism of $\mathbf{C}$ by applying it to B and the identity on B, say $f^\sharp = (\phi_B)(\mathrm{id}_B): B \to C$. This inverts * since $(f^*)^\sharp = (f^*_B)(\mathrm{id}_B) = \mathrm{id}_B \circ f = f$.

Now this is not just all theory. Let us take a very simple category with one object, and morphisms, lists of numbers with nil and append (this is just the usual free monoid construction). Suppose L, M, N are lists. We can define embed, i.e. *, by

$$\text{embed } l = \lambda m.\text{append}(l, m),$$

and extrude, i.e. $^\sharp$, by

$$\text{extrude } f = f \text{ nil};$$

put $F = \text{embed } L$, $G = \text{embed } M$ and $H = \text{embed } N$. Now instead of the "combinatory" expression

$$\text{append}(L, \text{append}(M, \text{append}(L, \text{append}(N, \text{nil}))))$$

we can write the "functional" expression

$$Kx = F(G(F(Hx)))$$

$$\text{extrude } K.$$

We can turn lists into functions and combine them with lambda calculus, or some syntactically sugared version of it. And we can do this with any category (provided we have polymorphic lambda calculus). Put

$$\text{embed } T\ U\ G = \lambda S.\lambda F.\text{comp } S\ T\ U\ F\ G$$

$$\text{extrude } T\ U\ G' = G'(\text{ident } T).$$

We use primed variables, such as G', for "embedded deliverables", that is polymorphic functions from deliverables to deliverables.

We can write our previous definition

$$\text{ssEE} = \text{comp sEO sOE doubleNE } = \text{Natrec Even zE ssEE}$$

as simply

$$\text{doubleNE}' = \text{Natrec}' \text{ Even zE}'(\lambda n.\text{sEO}(\text{sOE } n))$$

where Natrec' is derived from Natrec, but works on functions rather than morphisms (it uses embed and extrude)—it is a combinator over embedded deliverables. Now doubleNE = extrude $\text{doubleNE}'$.

Thus we can employ a notation for writing down deliverables which approximates to a functional notation which we would use for writing programs, rather than the more opaque combinatory one. We write down functional expressions for embedded deliverables. We have implemented the functions embed and extrude in LEGO.

The first author (RB) is grateful to Malcolm Bird (private communication) who showed him this technique some years ago for another example; at the time he did not realise that it was connected with the Yoneda embedding.

Second order deliverables

The system which we have described above amounts to a functional version of the well known invariants used in proofs of imperative programs. Unfortunately there is no connection between the input and the output of the function. All we say is that if the input is in set S then the output is in set T, but there is no *relation* between them. For example we might specify that a sorting function takes lists to ordered lists, but we cannot specify that the output is a permutation of the input. The function might always produce the empty list, which is indeed sorted, but not very interesting. As a matter of fact the classical invariant proofs have the same weakness, masked by a tacit assumption that some variable which is carried through the computation does not change its value. To enforce the constraint that the output list be a permutation of the input list we need to resort to "second order" deliverables, where the deliverables defined above are "first order" ones.

So let us write Del^1 for Del above and del^1 for del. We now define Del^2 and del^2. Suppose $S{:}\, s \to \text{Prop}$, $P{:}\, s \to p \to \text{Prop}$ and $R{:}\, s \to r \to \text{Prop}$ for types s, p and r. Let

$$\text{Del}^2_{s,p,r}\ S\ P\ R\ f = \forall x.Sx \to (\forall y.P\ x\ y \to R\ x\ (f\ x\ y))$$

that is $\forall x.Sx \to \text{Del}^1(P\,x)(R\,x)(f\,x)$. $\text{del}^2_{s,p,r}\ S\ P\ R$ is the dependent sum type $\Sigma f{:}\, p \to r.\text{Del}^2_{s,p,r}\ S\ P\ R\ f$. There is a category $\mathbf{C}_s S$ whose objects are (p, P), (r, R), ... for relations P, R, ... and whose morphisms from (p, P) to (r, R) are elements of $\text{del}^2_{s,p,r}\ S\ P\ R$.

If $P_1{:}\, s \to p_1 \to \text{Prop}$ and $P_2{:}\, s \to p_2 \to \text{Prop}$, then they have a product $P_{12}{:}\, s \to (p_1 \sharp p_2) \to \text{Prop}$ given by $P_{12}\ x\ y_{12} = S\ x \to (P_1\ x\ (\pi_1\ y_{12}) \wedge P_2\ x\ (\pi_2\ y_{12}))$, with projections $\Pi{:}\, P_{12} \to P_i$ given by $\Pi\ x\ y_{12} = (\pi_i\ y_{12}, \ldots)$ where $\ldots$ is the (trivial) proof that for all $x{:}\, s$, $Sx \to (\forall y_{12}{:}\, p_1 \sharp p_2, P_{12}\ x\ y_{12} \to P_i\ x\ (\pi_i\ y_{12}))$.

We conjecture that $\mathbf{C}_s S$ is cartesian closed (modulo the equivalences mentioned earlier) although we haven't checked the details.

Given $F\colon \mathrm{del}^1\ T\ S$ and $G\colon \mathrm{del}^2\ S\ P\ R$ we can define objects $(F^*\,P)$, $(F^*\,R)$ and substitution $F^*\,G\colon \mathrm{del}^2\ T\ (F^*\,P)\ (F^*\,R)$. We conjecture that this substitution operation defines an indexing of del^2 over del^1, similar indeed to the general categorical framework for dependent types. This suggests a reflection in which a dependent product type may be used to name a hom-set of deliverables (compare Martin-Löf's "subset interpretation" of type theory). We are currently investigating all this.

One could proceed in analogous manner to third order deliverables and so on, but it is not clear how useful these are. Since we have Σ-types we can presumably code these as second order deliverables (thanks to Gordon Plotkin for this remark).

As an example, we have been experimenting with these second order deliverables for the proof of an insert sort. First we proved an induction principle over sorted lists, which enables us to establish a relation between an input list and the result of applying a recursive function to it:

$$\text{for } \phi\colon (\text{list } \alpha) \to \beta \to \text{Prop}, \quad n\colon \beta, \quad c\colon \alpha \to (\text{list } \alpha) \to \beta \to \beta$$

$$\dfrac{\phi\ \text{nil}\ n \qquad \begin{array}{c}\left[\text{Sorted } (a{::}l),\ \ \phi\ l\ b\ \ [a\colon \alpha,\ l\colon \text{list } \alpha,\ b\colon \beta]\right] \\ \vdots \\ \phi\ \ (a{::}l)\ \ (c\ a\ l\ b)\end{array}}{\forall l\colon \text{list } \alpha.\ \ \text{Sorted } l \supset \phi\ l\ (\text{listrec } n\ c\ l)}\ \text{Sorted List Induction}$$

From this principle we obtain the following operation on deliverables, which involves a dependent family, F, of second order deliverables:

$$\dfrac{N \in \mathrm{del}^1\ \text{Unit nil}^*\,\phi \quad F \in \Pi a\colon \alpha.\mathrm{del}^2\ ((\text{cons } a)^*\ \text{Sorted})\quad \phi \quad (\text{cons } a)^*\,\phi}{(\text{listrec } n\ c, \ldots) \in \mathrm{del}^2\ \text{Sorted True } \phi}$$

where $n = N.1$ and $c = \lambda a\colon \alpha.((F\ a).1)$. In the proof of insert sort, we have applied this construction to the invariant $\phi = \lambda l, m\colon \text{list } \alpha.\text{Sorted } m \wedge \text{Perm } l\ m$. This enables us to prove that the output of sort is both sorted and a permutation of the input.

References

Constable R. *et al.* (1986), Implementing Mathematics with the NuPrl Proof Development System, Prentice Hall.

Coquand T. and Huet G. (1988), "The Calculus of Constructions", Information and Computation, 76.

Lambek J. and Scott P. J. (1986), Introduction to Higher Order Categorical Logic, Cambridge University Press.

Luo Z., Pollack R. A. and Taylor P. (1989), LEGO User Manual, LFCS, Dept. of Computer Science, Edinburgh University, UK.

Nordström B., Peterssen K. and Smith J. (1990), Programming in Martin-Löf Type Theory, Oxford University Press.

Paulin-Mohring C. (1989), Extracting F_ω's Programs from Proofs in the Calculus of Constructions, Proc. Conf. on Principles of Programming Languages, ACM.

A plea for weaker frameworks

N.G. de Bruijn

Eindhoven University of Technology

Abstract

It is to be expected that logical frameworks will become more and more important in the near future, since they can set the stage for an integrated treatment of verification systems for large areas of the mathematical sciences (which may contain logic, mathematics, and mathematical constructions in general, such as computer software and computer hardware). It seems that the moment has come to try to get to some kind of a unification of the various systems that have been proposed.

Over the years there has been the tendency to strengthen the frameworks by rules that enrich the notion of definitional equality, thus causing impurities in the backbones of those frameworks: the typed lambda calculi. In this paper a plea is made for going into the opposite direction: expel those impurities from the framework to the books, where the role of definitional equality is taken over by (possibly strong) book equality.

1 Introduction

1.1 Verification systems

A verification system consists of

(i) a framework, to be called the *frame*, which defines how mathematical material (in the wide sense) can be written in the form of *books*, such that the correctness of those books is decidable by means of an algorithm (the *checker*),

(ii) a set of basic rules (*axioms*) that the user of the frame can proclaim in his books as a general basis for further work.

For several reasons (theoretical, practical, educational and sociological) it may be recommended to keep the frames as simple as possible and as weak as possible, and to prefer frames belonging to a hierarchy in which the various specima can be easily compared.

We shall restrict the discussion to frames that have typed lambda calculi as their backbone, an idea that is gaining field today, along with the rising popularity of the principle to treat proofs and similar constructions in exactly the same way as the usual mathematical objects. The rules of the frame involve rules for handling lambda terms, lambda reductions, and rules about attaching a type to a term. The typing rules are strongly influenced by the notion of *definitional equality.* The latter notion is primarily based on the reductions of the lambda calculus.

1.2 Strengthening the frame

Over the years there has been the tendency to put more and more power into frame, at the price of impurities of the typed lambda calculus. This already started with the early versions of Automath (AUT68 and AUT-QE, see [3], [4], [13]) which were very advanced for their time (around 1968) in the sense that the frames were light and that almost all logical and mathematical knowledge was to be developed in the books, but nevertheless there was the impurity of so-called type inclusion (see section 4.3). A proposal to get rid of it (see [11]) came too late for implementation in the Automath project).

1.3 Motives for strengthening

Let us list some of the motives that one might have for strengthening the frame.

(i) One has the idea that the rules of the frame can be handled automatically and that book material has to be guided essentially by hand. This explains that wherever one sees things that can be automated, one has the tendency to shift them from the book into the frame.

(ii) Mathematicians have been trained to base their work on as few axioms as possible, preferring strong axioms over lists of weaker ones. So much the nicer if one can shift axioms into the frame, where one does not see them any more!

(iii) Strengthening the frame can enlighten the burden of writing books.

(iv) It is natural to have the feeling that the way one thinks about mathematics and its implementation in a verification system should somehow be reflected in the frame.

In particular these points may mean that much of what is achieved by book equality (the kind of equality that is to be considered as a mathematical proposition and for which proofs have to be provided by the user of the system) can be shifted to definitional equality, which is based on reductions in the frame that can be handled automatically).

1.4 Candidates for admission into a frame

There are two kinds of candidates, the innocent ones and the others. The innocent ones are what one might call *syntactic sugar*. They deal with ways to abbreviate book material, usually in a way that corresponds to existing habits of mathematicians. One might decide to enrich the frame with rules for such forms of sugaring.

Under the heading of sugaring we have things like:

(i) Ways to handle finite strings of terms, and strings of applicators and abstractors (in the latter case we call them *telescopes*, see [17]) by means of a single identifier, just as if they were ordinary objects.

(ii) Ways to to omit pieces of input and output that can be retraced easily and uniquely by the checker.

(iii) In particular this may refer to ways to replace explicit proofs and similar constructions by hints.

Sugaring is innocent since it has no influence on the validity and the interpretation of the things we derive in our verification system.

The other candidates are not necessarily innocent. They may stem from mathematical or logical insights, and once these have been incorporated in the frame, they may get applications they were not originally intended for. We might even fear antinomies.

Unlike the situation in a law court, the accused is to be considered as guilty as long as its innocence has not been established. This may mean that strengthening the frame gives us quite some work, possibly of a model-theoretical nature.

The non-innocent ways to strengthen the frame all increase the number of valid definitional equalities and the number of valid typings. This does not necessarily mean increasing the number of derivable mathematical results, but it might. And some of these extensions (like (iv)) damage the purity of the typed lambda calculus backbone.

We mention

(iv) Rules for type inclusion, or, what amounts to the same thing, Π-rules (see section 4.3).

(v) The principle of *proof irrelevance* (cf. [13], [21]), declaring two proofs for one and the same proposition to be definitionally equal.

(vi) Shifting knowledge about natural numbers, induction and recursion into the frame.

1.5 A plea for weaker frames

When one uses arguments of economy in order to put more power into the frame, one forgets that it is neither very hard to put some automation in the task of writing books. We can invent abbreviational facilities that may result in automatic writing of parts of the books (parts that the user even does not need to see if he does not want to). In particular this may refer to handling trivial equalities like those we need when dealing with pairs and cartesian products. This need for automatic textwriting will turn up in many other situations too. It happens every now and then, when we want to economize on writing things that we consider as part of our subconcious thinking. It does not seem to be the right thing to enrich the frame at every occasion of that kind.

And needless to say the tendency to provide the frame with facilities that depend on local and personal preferences may lead to an undesirable variety of diverging frames.

Let us list a number of arguments for retreating to weaker frames.

(i) Weaker frames means fewer frames. We are still far from a general acceptance of typed lambda calulus verification systems in the scientific world, the mathematical world in particular. Maintaining a variety of competing but hardly compatible systems will not be very helpful in this respect. And it will neither be very helpful for establishing cooperation between the various groups handling verification systems.

(ii) Having to understand a verification system, one has to see the clear borderline between frame and books. The distinction is easy to understand if the frame contains everything that can not be treated in the books, and nothing else.

(iii) For many people a pure system of typed lambda calculus is already hard enough. Impurities make it harder.

(iv) One should not think that the frame is the only place where things can be done automatically. There are plenty of cases for automated or semi-automated book-writing anyway. Verification systems would hardly have a future if we would have to depend on doing all the tedious and dirty work by hand.

(v) There is no reason at all to let any kind of syntactic sugaring slip from the book into the frame. Sugaring is meant to make reading and writing easier, or better adapted to traditional notations, and that belongs entirely in the world of the book. The frame is supposed to be operated by machines which do not have the same ideas as humans have if it comes to the question whether things are difficult or tedious to handle.

(vi) The pain of having to keep things book-equal that one would prefer to have definitionally equal can be somewhat relieved by the introduction of a third kind of equality: *strong book equality* (see section 5.8).

1.6 Hierarchy of frames

In section 1.5 it was recommended to limit the number of frames that we present to the world. But the world would certainly also appreciate seeing a clear hierarchical structure among the different frames we present. The hierarchical relation $F_1 \leq F_2$ should be the one that expresses that every book valid with respect to frame F_1 is also valid with respect to F_2.

The hierarchy need not necessarily be linear, but it would certainly be appreciated if the hierarchy contains a maximal element F_m with $F \leq F_m$ for all the F in the hierarchy.

The inequality $F_1 \leq F_2$ has to be kept more or less informal in the present discussion. After all, we have not given a mathematical definition of the notion "frame". We shall also use the notation $F_1 \leq F_2$ in cases where the notion of a book valid with respect to a frame still needs some interpretation (like in the case of $\Delta\Lambda$, where the system itself does not say how a book has to be considered as a lambda term).

1.7 Implementability

A still more informal notion is the one of implementability. We write $F_1 \xrightarrow{\mathrm{i}} F_2$ if F_1 can be *implemented* in F_2. By this we mean that the essence of what F_1 can do for us can also be done in the (weaker) system F_2, provided that the books written under F_2 start with a number of suitable axioms, and that we take it for granted that some of the definitional equalities in F_1 are to be replaced by book equalities.

We should not forget that $\xrightarrow{\mathrm{i}}$ indicates a rather weak form of implementation; one might rather call it *mimicking*. Nevertheless it might give an idea about the strength of various systems, both in a theoretical and a practical sense.

1.8 Purity of lambda calculi

We shall be informal here, *not* start by saying what a typed lambda calculus is. We just stress the essential items for purity.

(i) Definitional equality ($\stackrel{\mathrm{d}}{=}$) should be defined by means of β-reduction only. It would probably do no harm to admit η-reduction too, but it seems hardly worth while. We do not mention α-reduction since its role can be reduced to the level of syntactic sugaring; one can assume that the frame deals with namefree lamda calculus.

(ii) We should require full unicity of types. So assuming the typing $P : Q$ and the validity of R, we have $P : R$ if and only if $Q \stackrel{\mathrm{d}}{=} R$.

(iii) We should require that both abstraction and application commute with typing. So with the usual notation

$$\frac{\Gamma, (x:A) \ \vdash \ P(x) : Q(x)}{\Gamma \vdash \ [x:A]P(x) : [x:A]Q(x)},$$

$$\frac{\Gamma \vdash \ P : Q \quad \Gamma \vdash \ \langle a \rangle Q}{\Gamma \vdash \ \langle a \rangle P : \langle a \rangle Q}.$$

Here we followed the Automath notation for abstraction and application. Abstractors are written like $[x : A]$, where x is the bound variable and A its type, and abstractors are written in front of the term they act on. Applicators are also written in front: $\langle a \rangle P$ describes what is interpretated as a function value, where P is the function and a the argument.

The above rule for application is not sufficient for getting all cases of $\Gamma \vdash \langle a \rangle Q$. We have to add the rule

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash \ Q \stackrel{\mathrm{d}}{=} [x:A]R(x)}{\Gamma \vdash \ \langle a \rangle Q}.$$

2 A proposal for a hierarchy of frames

We shall discuss three frames, forming a linear hierarchy:

$$\text{AUT-QE-NTI} \ \leq \ \Delta\Lambda \ \leq \ \Lambda\Sigma.$$

The first one, AUT-QE-NTI , was designed in 1978 as an Automath language without impurities. The other members of the Automath family can be implemented in it (in the sense of section 1.7 (see [11])). In particular we have

$$\text{PAL} \ \leq \ \text{AUT68} \ \leq \ \text{AUT-QE} \ \stackrel{\mathrm{i}}{\rightarrow} \ \text{AUT-QE-NTI},$$
$$\text{AUT}\Pi \ \stackrel{\mathrm{i}}{\rightarrow} \ \text{AUT-QE-NTI}.$$

For PAL, AUT68 and AUT-QE we refer to [3], [4], [13].

$\Delta\Lambda$ was designed in 1985. It is very close to the system Λ for which Nederpelt proved strong normalization ([20]). That Λ was a reformulation of the system AUT-SL ([5]) which presented complete Automath books (with extremely liberal abstraction rules) in the form of a single line. In order to do that, all definitions of the book had to be eliminated first, and therefore AUT-SL had theoretical interest only. With $\Delta\Lambda$, however, this elimination of definitions is no longer necessary (see section 5.2).

$\Delta\Lambda$ is strong enough for implementation (in the sense of $\stackrel{\mathrm{i}}{\rightarrow}$) of Barendregt's Generalized type systems (see [1]), which in particular contains the Coquand-Huet theory of constructions. It may be interesting to compare Barendregt's [1] with the much older note [7]. The essential differences are (i) in [7] type inclusion (cf. section 4.3 below) was restricted to degree 1, and (ii) in [1] there is no place for the typical Automath feature of admitting more contexts than those for which it is allowed to abstract from.

In section 6 we shall devote some attention to the system $\Lambda\Sigma$, which is intended as a very substantial extension of typed lambda calculus. But at present it should be said that the relation $\Delta\Lambda \leq \Lambda\Sigma$ is rather a program for development of $\Lambda\Sigma$ than a fact.

3 The typed lambda calculus $\Delta\Lambda$

3.1 Syntax

The difference between $\Delta\Lambda$ and Nederpelt's Λ lies in the notion of correctness, not in the syntax.

Let us agree that what we really mean is the namefree version, and that the use of names of variables is syntactic sugar (both in the description as in the use of the syntax). A complete description was given in [15] by means of trees with reference arrows (arrows from the variables at end-points of the tree to the corresponding lambda's). That reference arrow system seems to be the easiest way to describe syntax and language theory. At the stage of implementation in a checker one might decide how to implement the arrows. One might use depth references, but if the implementation is in terms of a programming language with a pointer mechanism, one might use those pointers instead. For input and output of mathematical texts, however, there is much to be said for the syntactic sugar of named variables.

In order to avoid repeating the complete formal description given in [15] we give an informal one here, with the use of names of variables. In the first round of syntax description we formulate

(i) τ is a term.

(ii) A variable is a term.

(iii) If x is a variable, if P and Q are terms, where P does not contain x, then $[x : P]Q$ is a term.

(iv) If P and Q are terms, then $\langle P\rangle Q$ is a term.

In the second round we add the restriction that a term should not contain any free variables.

Every bound variable in a term has a unique type, obtained from (iii) by agreeing that all x occurring in Q get type P.

Every term has a terminating symbol on the extreme right. If that symbol is a τ we say that the term has degree 1, and we do not define the type of the term. If the terminating symbol of the term E is a variable x, we get the *type* of E (notation $\mathrm{typ}(E)$) if we simply replace that terminal occurrence of x by the type of x.

And we define the *degree* of a term E recursively by $\mathrm{degree}(E) = \mathrm{degree}(\mathrm{typ}(E)) + 1$.

3.2 Correctness

Correctness of terms depends on the notions of typing and definitional equality.

Definitional equality in $\Delta\Lambda$ is defined by means of *mini-reductions*: local β-reductions and so-called AT-removals, instead of by the global β-reduction used in Λ. Local β-reduction means that some sub-term $\langle R\rangle[x : P]Q$ is transformed into $\langle R\rangle[x : P]Q^*$, where Q^* is obtained from Q upon replacing a single one of the occurrences of x by R. It can of course be done only if there is at least one occurrence of x in Q. If there are no occurrences of x in Q we can apply AT-removal: we remove the AT-pair $\langle R\rangle[x : P]$, so $\langle R\rangle[x : P]Q$ turns into Q. (A few words to explain the "AT": in the metalanguage used in [15] the "A" refers to applicators $\langle\ \rangle$, and the "T" to typed abstractors $[\ :\ ]$.)

In the usual way we get the notion $\stackrel{\mathrm{d}}{=}$ of definitional equality by reflexive transitive symmetric closure of the set of mini-reductions.

For practical reasons it can be recommended to extend the set of mini-reductions by basing them on AT-*couples* instead of AT-pairs (see [15], section 4.3). As an example we mention that in $\langle R\rangle\langle Q\rangle\langle P\rangle[x:A][y:B][z:C]$ the $\langle R\rangle$ and $[z:C]$ form an AT-couple. This extension of the notion of mini-reductions does not extend the notion of $\stackrel{\mathrm{d}}{=}$.

Having the notion $\stackrel{\mathrm{d}}{=}$, we can sketch the matter of correctness.

The essential point in the definition of correctness is in the situation of subterms of the form $\langle R\rangle P$. The usual conditions in typed lambda calculi, in particular in Nederpelt's Λ, are

(i) R and P are both correct,

(ii) in the finite sequence P, $\mathrm{typ}(P)$, $\mathrm{typ}(\mathrm{typ}(P))$, ... there is a term that is definitionally equal to a term of the form $[x:S]W$, with $\mathrm{typ}(R) \stackrel{\mathrm{d}}{=} S$.

In $\Delta\Lambda$ the definition of correctness is arranged differently, in an algorithmic way. The algorithm can be described in two rounds. In the first round the algorithm runs through the term from left to right, and for every applicator it produces a pair of terms of which the definitional equality remains to be checked. The first round can be completed irrespective of whether all these definitional equivalences can be established or not. And we mention that this first round runs in linear time. In the second round the definitional equivalences of the list have to be checked. This second round does not depend on typing any more. It can be interpreted in untyped lambda calculus, and it is a matter of practice to devise an efficient strategy. It is certainly a poor strategy to try to establish $\stackrel{\mathrm{d}}{=}$ by evaluating normal forms: these can be exceedingly long.

If a term is correct in the sense of Λ then it is also correct in the sense of $\Delta\Lambda$, but the converse is not true. $\Delta\Lambda$ is more liberal concerning the applications $\langle R\rangle P$. In $\Delta\Lambda$ it is no longer required that P is correct all by itself: the correctness check may make use of the fact that $\langle R\rangle$ is in front of this P.

We shall see in section 5.2 how this feature enables us to interpret an Automath book as a term in $\Delta\Lambda$. The algorithm for checking the correctness of the term is an efficient algorithm for checking the correctness of the whole book.

4 Characteristics of Automath languages

4.1 Generalities

Without trying to explaing much in detail, we just indicate a number of issues in which Automath differs from some other verification systems based on typed lambda calculi.

Automath books are written as sequences of lines: definitional lines and primitive lines. Every line is written in a certain context, which is a sequence of typed variables

$$(x_1 : A_1)\cdots(x_k : A_k),$$

where each A_j is a term that may contain the previous $x_1,\cdots,x_{j-1}$. As meta-notation for contexts we use symbols like Γ, and for conxtext extensions $\Gamma,(x:A)$. There is a notion of validity of contexts, a notion of validity of terms (lambda-typed lambda terms) inside a context (meta-notation $\Gamma\vdash P$), and a notion of validity of typings inside a context (meta-notation $\Gamma\vdash P:Q$).

A definitional line (in context Γ) has the form "$f \ := \ P \ : \ Q$", where f is a new identifier, and where $\Gamma \ \vdash \ P \ : Q$. A primitive line has the same form, but for the fact that the term P is replaced by the symbol PN (which is not considered as a term).

There is a fixed set of *basic* terms (like τ, **type**, **prop**) which are said to have *degree* 1. The degree of all further valid terms is found by the rule that if $P : Q$ then degree(P) = degree(Q) + 1. And if $f := P : Q$ (or $f :=$ PN : Q) is a line then degree(f) = degree(Q) + 1. In Automath the degrees are restricted to the values 1, 2, 3, so in lines $f := P : Q$ or $f :=$ PN : Q the Q should have degree 1 or 2.

Abstractors and applicators are denoted as in section 1.8.

Some of the Automath languages (AUT-QE , AUT-QE-NTI) admit *quasi-expressions*, which have the form of a sequene of abstractors followed by a basic term, like

$$[x_1 : A_1] \cdots [x_k : A_k] \ \mathbf{type}.$$

They are given degree 1.

4.2 Instantiation

Very typical for Automath is the fact that validity of contexts is more liberal than the rules for abstraction. Abstractors [x:**type**] are not allowed, but context extensions (x:**type**) are. A valid context Γ can be extended to a new valid context by any extension $\Gamma, (x : Q)$ provided that x is a new variable, that $\Gamma \vdash Q$ and that the degree of Q is 1 or 2. The requirements for abstraction can be more severe. Abstraction $[x : A]$ is restricted anyway to the case that A has degree 2, in AUT68 and in AUT-QE-NTI moreover by the condition that the type of A is a basic term. In PAL there are no abstractors at all.

In the Automath languages there are (like in standard mathematical language) two different devices for describing functional relationship, the *instantiation device* and the *lambda device*. The first one admits explicitly defined functions. If inside a context

$$\Gamma, \ (x_1 : A_1) \cdots (x_k : A_k)$$

we have a line $f := P : Q$ or $f :=$ PN : Q then in any later line (in a context Γ_1 that at least starts with Γ), we can *instantiate* f by attaching a string of sub-terms, writing $f(B_1, \ldots, B_k)$. These B_j have to be typed (in the context Γ_1) by the things we get from the corresponding A_j if we replace the x_i's by the corresponding B_i's.

In cases where $[x_1 : A_1] \cdots [x_k : A_k]$ are admissible abstractors, the step from f to $f(B_1, \ldots, B_k)$ corresponds to the one from F (where F is obtained from $f(x_1, \ldots, x_k)$ by k-fold lambda abstraction) to the k-fold application $\langle B_k \rangle \cdots \langle B_1 \rangle F$. It has to be mentionend that the instantiation device generates new definitional equalities by means of the δ-reduction, which is essentially nothing but replacing the defined identifier f by its definition P (in case the line was $f := P : Q$).

In the instantiation device the notion of function is restricted to explicitely defined functions, so we cannot describe mathematics beyond the level of the 18-th century. Nevertheless the system of "proofs as objects" enables the lambda-free Automath language PAL to use the instantiation device for true mathematical reasoning.

But what the instantiation device cannot do is to let us talk about arbitrary functions, and to *say* in the language that a function obtained by the instantiation device is a function indeed. That is what we need the lambda calculus for. We might say that the lambda calculus internalizes instantiation.

4.3 Type inclusion

Type inclusion is the point where the Automath languages AUT68 and AUT-QE deviate from pure lambda calculus. Having type inclusion is equivalent to having Π-rules like those in Zucker's AUT-Π (see [21]) and in many other systems of typed lambda calculi. Actually one can say that the notations with Π serve as sign posts for places where type inclusion has been applied. The checker never needs those indications, since it is always directly retraceable where and how type inclusion has been applied. That retraceability was the main reason why such indications were omitted in the design of Automath.

The meta-language of Π-rules seems to be more generally accepted than the one of type inclusion, and the feature of type inclusion may have been one of the reasons why in the early days of Automath logicians were inclined to dislike it. Therefore it may be the right place here to give it some attention, not just to the formal rules but also to interpretations.

The original idea about using a notation $Q : \mathbf{type}$ was that it just served as an indication that Q had degree 2, what meant that Q could be used as a type for other terms, like in $P : Q$. There is an obvious analogy between the "real" typing $P : Q$ and the notation $Q : \mathbf{type}$. They play the same role in the formation of contexts: by context extensions $(Q : \mathbf{type})$ we can introduce type variables. And they play the same role in definitional lines: just like such lines can be used for defining new objects (of degree 3), they can be used for defining types (of degree 2). And primitive lines can be used for creating primitive types (of degree 2) just like they can be used for creating primitive objects (of degree 3). All this can be achieved without saying that **type** is a term; instead of considering $Q : \mathbf{type}$ as a relation between two terms, we consider the combination "$: \mathbf{type}$" as a kind of predicate applied to Q.

In the light of this opinion about **type** one has to interpret the AUT68 rule

$$\frac{A : \mathbf{type} \qquad \Gamma, (x : A) \vdash Q(x) : \mathbf{type}}{\Gamma \vdash [x : A]Q(x) : \mathbf{type}}.$$

It just says that $[x : A]Q(x)$ is again a valid term of degree 2.

The next step is that sometimes we we want to say more about $[x : A]Q(x)$ than that it has degree 2. We sometimes want to express about some term H of degree 2 that it behaves like a term that starts with the abstractor $[x : A]$. If we have some $a : A$ we may wish to form the application $\langle a \rangle H$, and if moreover we have some $f : H$ we may want to form $\langle a \rangle f$. In order to register those possibilities, the *mock typing* $H : [x : A]\mathbf{type}$ was invented, and that extended AUT68 into AUT-QE. Note that the acronym QE in AUT-QE stands for "quasi-expressions", clearly indicating that $[x : A]\mathbf{type}$ was not considered as a term. As before, the combination "$: [x : A]\mathbf{type}$" was considered as a kind of predicate applied to H.

The mock typing $H : [x : A]\mathbf{type}$ just gave more information about H than just $H : \mathbf{type}$.Of course we took the right in AUT-QE to pass from $H : [x : A]\mathbf{type}$ to $H : \mathbf{type}$ (but not the other way round). It can be interpreted as sacrifice of information. We can see it as applying an inclusion $[x : A]\mathbf{type} \subset \mathbf{type}$, and accordingly the step from $H : [x : A]\mathbf{type}$ to $H : \mathbf{type}$ was called *type inclusion*. It can also be applied repeatedly:

$$[x : A][y : B][z : C]\mathbf{type} \subset [x : A][y : B]\mathbf{type} \subset [x : A]\mathbf{type} \subset \mathbf{type}$$

(note that B may depend on x, and C on both x and y).

Once we start implementing AUT-QE in a lambda calculus like $\Delta\Lambda$ (cf. section 5.3) we of course begin to consider all the quasi-expressions as lambda terms, and the mock typings as typings of the typed lambda calculus, and that means having impurities. The type inclusion rule conflicts with the idea of unicity of types.

For dicussions where the type inclusion feature is compared to the usage of Π in AUT-Π we refer to [18] and [10].

5 AUT-QE-NTI

5.1 The rules

The syntax of AUT-QE-NTI is the same as the one of AUT-QE. The main difference is type inclusion: the acronym NTI stand for "no type inclusion".

AUT-QE-NTI handles quasi-expressions (see section 4.3). And all quasi-expressions can be used for typing of variables in context extensions. The general rule is that whenever Γ is a valid context, and $\Gamma \vdash Q$, where the degree of Q is 1 or 2, then $\Gamma, (x : Q)$ (with a new variable x) is a valid context. But where AUT-QE allows abstraction over all Q with degree 2, AUT-QE-NTI restricts it to the case where $Q : B$, where B is a basic term (like **type**, **prop**, or τ). So the abstraction rule is

$$\frac{\Gamma \ \vdash \ A : \mu, \quad \Gamma, (x : A) \ \vdash \ P(x) : Q(x)}{\Gamma \vdash \ [x : A]P(x) : [x : A]Q(x)},$$

where μ is one of the basic terms. The rules for application and instantiation are the same as for AUT-QE.

In our further discussions about AUT-QE-NTI we shall ignore the possibility to have more than one basic term, and just formulate everything for τ only.

5.2 AUT-QE-NTI books as terms in $\Delta\Lambda$

We shall explain here how a book written in AUT-QE-NTI can be transformed into a single term of $\Delta\Lambda$. The book consists of a finite sequence of lines; some are primitive lines, others are definitional lines. Corresponding to every primitive line we form a corresponding abstractor, and to every definitional line a pair consisting of an applicator and an abstractor. We just concatenate these abstractors and applicator-abstractor pairs in the order of the lines of the book, and we complete that sequence with a τ. That concatenation will be the term that represents the whole book.

A thing that should be looked into first, is that $\Delta\Lambda$ does not handle instantiation, and therefore the instantiations in the AUT-QE-NTI book have to be remodelled to applications. In the last line of the example given below there is a (double) case of instantiation, and from that example it will be clear how the remodelling works in general.

In [6] a notational convention was explained that can be helpful here. If the identifier f is introduced in a context of length k, then the identifier ③f will be used for what we get from f by abstracting over the last 3 pieces of the context, and the reader is suppposed to understand this without seeing a line where that new identifier ③f was introduced. With this convention we can say that the instantiation $f(U, V, W)$ is definitionally equal to the application $\langle W\rangle\langle V\rangle\langle U\rangle$ ③f.

We now describe how the abstractors and applicator-abstractor pairs are obtained from the book lines. In the case of a primitive line

$$(x_1 : A_1) \cdots (x_k : A_k) \ \vdash \ f \ := \ \mathrm{PN} \ : \ Q$$

we create a new identifier F and take as the abstractor

$$[\ F \ : \ [x_1 : A_1] \cdots [x_k : A_k] Q\ ];$$

in the case of a definitional line

$$(x_1 : A_1) \cdots (x_k : A_k) \ \vdash \ f \ := \ P \ : \ Q$$

we form the applicator-abstractor pair

$$\langle [x_1 : A_1] \cdots [x_k : A_k] P \rangle \ [\ F \ : \ [x_1 : A_1] \cdots [x_k : A_k] Q\ ].$$

The new identifier F should not be confused with the old f (with the notation mentioned above it should be ③f, only if the context was empty there is no danger for such confusion.

We give a short example. Let the book be

	$\vdash$	A	$:=$	PN	$:$	τ
	$\vdash$	c	$:=$	PN	$:$	A
	$\vdash$	f	$:=$	$[y : A]c$	$:$	$[y : A]A$
$(x : A)$	$\vdash$	g	$:=$	$\langle x \rangle f$	$:$	A
	$\vdash$	B	$:=$	A	$:$	τ
$(z : \tau)$	$\vdash$	h	$:=$	$g(g(c))$	$:$	B

This book transforms into the following term in $\Delta\Lambda$:

$$[A : \tau] \ [c : A] \ \langle [y : A]c \rangle \ [f : [y : A]A] \ \langle [x : A] \langle x \rangle f \rangle \ [G : [x : A]A]$$

$$\langle A \rangle \ [B : \tau] \ \langle [z : \tau] \langle \langle c \rangle G \rangle G \rangle \ [H : [z : \tau]B] \ \tau.$$

Under this translation from a book in AUT-QE-NTI to a term in $\Delta\Lambda$ we see how essential the difference between $\Delta\Lambda$ and Λ (indicated at the end of sectio 3.2) is. If in an AUT-QE-NTI book we replace some definitional line $f := P : Q$ by the primitive line $f := \mathrm{PN} : Q$, the rest of the book will often become incorrect, since the validity of the book might have depended on the definitional equality of f and P, which gets lost if we replace P by PN. But if in the definition of correctness in $\Delta\Lambda$ we would have insisted that correctness of $\langle R \rangle P$ requires correctness of P (see section 3.2), we would actually have required that the term in $\Delta\Lambda$ that corresponds to the AUT-QE-NTI book would remain correct after just omitting the applicator that carries the information about the definition of f, and that means the same thing as replacing the P by PN.

The example above demonstrates this for the pair $\langle A \rangle \ [B : \tau]$ If we would omit the $\langle A \rangle$, the type of $[z : \tau] \langle \langle c \rangle G \rangle G$ would no longer be $[z : \tau]B$.

In AUT-SL (see [5]) there was the standard convention that in an application $\langle R \rangle P$ both parts P and R were required to be correct, and that made it necessary to eliminate all definitions of the book by δ-reductions before the translation to a lambda-term could begin. For that reason

AUT-SL could not be more than a way to streamline Automath language theory, whereas $\Delta\Lambda$ is more than that: it is also a practical tool for working with Automath, in particular for the checker.

It is not too easy to say exactly what kind of terms in $\Delta\Lambda$ correspond to an AUT-QE-NTI book according to our translation, since inside a line the notion of correctness handled the old requirement about correctness of $\langle R\rangle P$. Here we might recommend the Procrustes technique: if a feature of AUT-QE-NTI does not fit in the bed $\Delta\Lambda$ then we just *make* it fit by changing the definition of AUT-QE-NTI. What we gain is the facility of admitting abbreviations which are local inside a line, or even inside a part of a term in that line. But a more important gain is the simplicity of the properties of the result of the embedding.

5.3 Implementing AUT-QE in AUT-QE-NTI

5.4 Introduction

AUT-QE-NTI is definitely weaker than AUT-QE and also weaker than AUT68, just because of its purity. In order to let AUT-QE-NTI enjoy the blessings of type inclusion without actually having it, we can start our book with a set of axioms that mimick the effects of type inclusion. In that way we can say that we can implement AUT-QE and AUT68 in AUT-QE-NTI.

There are two prices we have to pay:

(i) applications of the equivalent of type inclusion have to be made explicit in the book by means of references to those axioms, with explicit statement of the terms that have to be substituted for the parameters in these axioms,

(ii) the definitional equalities generated by type inclusion have to be replaced by book equalities.

We can try to overcome these objections by (i) providing automatic text writing facilities that does all the dirty work for us, and (ii) by distinguishing two different kinds of book equalities, a strong and a weak one, where the strong one can enjoy the decidability that the definitional equality is supposed to have (see section 5.8).

5.5 The axioms for Π

We now get to the axioms that mimick the type inclusion. Since they of course also mimick the role of Π, we shall use the letter Π as an identifier. Actually there is a multitude of axioms for Π, since the axioms get two parameters σ and μ that stand for basic types (like **type** or **prop**). We could do with a single set of axioms if we would handle only one basic term, or if we would introduce a facility for creating basic terms (that can be done by opening the possibility of degree 0, with a single term $*$ in it, and letting $\mu : *$ mean: "Let μ be a basic term").

As a set of axioms for Π we present

$(X:\sigma)\ (Q:[x:X]\mu)$	$\vdash\ \Pi_{\sigma\mu}$	$:=$	PN	$:\ \mu$
$(X:\sigma)\ (Q:[x:X]\mu)\ (u:\Pi_{\sigma\mu})$	$\vdash\ Ax1_{\sigma\mu}$	$:=$	PN	$:\ [x:X]\langle x\rangle Q$
$(X:\sigma)\ (Q:[x:X]\mu)\ (v:[x:X]\langle x\rangle Q)$	$\vdash\ Ax2_{\sigma\mu}$	$:=$	PN	$:\ \Pi_{\sigma\mu}$

These $Ax1$ and $Ax2$ permit us to jump up and down from Q to $\Pi(X,Q)$, and that mimics the type inclusion feature without frustrating the typing rules of the lambda calculus.

We remark that in two cases we have used the "η-expansion" $[x:X]\langle x\rangle Q$ instead of Q itself. This deviates from [11], and is done here in order to avoid having to apply η-reduction in cases of multiple application of the axioms for Π.

If one wants to implement mathematics without ever having to use η-reduction, it probably suffices to pass at once to the η-expansion of Q in all those cases where $Q:[x:X]\mu$ and Q is *not* definitionally equal to a term starting with the abstractor $[x:X]$ (this happens in cases like the one above, where Q was introduced as a variable typed by $[x:X]\mu$). This remark is similar to the observation by D. van Daalen, mentioned in the discussion on η-reduction in section 4.1.1 of [19].

We have to add equality axioms, expressing that we have (in abbreviated form)

$$Ax1(Ax2(v)) \;=\; v, \quad Ax2(Ax1(u)) \;=\; u.$$

The equality here will have to be book equality.

5.6 Axioms for Σ

Another set of axioms in AUT-QE-NTI can organize the operation Σ of AUT-Π. In the context $X:\mathbf{type}$, $Q:[x:X]\mathbf{type}$ it postulates the type $\Sigma(X,Q)$, that can be imagined as being the type of all pairs p,q with $p:X$, $q:\langle p\rangle Q$. With the terminology of telescopes (see [17] it can be described as a type that enables us to replace the telescope $[x:X][y:\langle x\rangle Q]$ of length 2 by the telescope $[w:\Sigma(X,Q)]$ of length 1. Actually this set of primitives was used extensively in AUT68 under the name "OwnType" in order to treat sets (subtypes) as types.

By multiple application of Σ we can condense telescopes of length >2 into length 1.

We can introduce Σ by means of the following axioms:

$(X:\sigma)\ (Q:[x:X]\mu)$	$\vdash\ \Sigma_{\sigma\mu}$	$:=$	PN	$:\ \mu$
$(X:\sigma)\ (Q:[x:X]\mu)\ (\varphi:\Sigma_{\sigma\mu})$	$\vdash\ \mathrm{proj1}_{\sigma\mu}$	$:=$	PN	$:\ A$
$(X:\sigma)\ (Q:[x:X]\mu)\ (\varphi:\Sigma_{\sigma\mu})$	$\vdash\ \mathrm{proj2}_{\sigma\mu}$	$:=$	PN	$:\ \langle\mathrm{proj1}_{\sigma\mu}\rangle Q$
$(X:\sigma)\ (Q:[x:X]\mu)\ (t_1:X)\ (t_2:\langle t_1\rangle Q)$	$\vdash\ \mathrm{pair}_{\sigma\mu}$	$:=$	PN	$:\ \Pi_{\sigma\mu}$

We have to add equality axioms, expressing (with book equality) that

$$\mathrm{proj}_1(\mathrm{pair}(t_1,t_2)) \;=\; t_1, \quad \mathrm{proj}_2(\mathrm{pair}(t_1,t_2)) \;=\; t_2,$$

$$\mathrm{pair}(\mathrm{proj}_1(\varphi),\mathrm{proj}_2(\varphi) \;=\; \varphi.$$

5.7 Further axioms

Needless to say we want some more axioms for dealing with standard mathematics, in particular axioms about book equality (a survey of what we need in AUT-QE can be found in [19]).

A thing that one might also like to implement is *proof irrelevance.* Proof irrelevance has been proposed as a rule for making different proofs of one and the same proposition definitionally equal (connected with the fact that mathematicians have the opinion that objects do not really depend on proofs), but that means another impurity of the typed lambda calculus. In the vein of AUT-QE-NTI it of course requires some book axioms, and the equality involved in it is a serious candidate for strong book equality (see section 5.8).

A further candidate for being shifted from the frame to book axioms is the matter of admitting more than 3 degrees. It is to be expected that this feature can be mimicked efficiently by means of axioms over a frame that handles only 3 degrees.

5.8 Three kinds of equality

The systems we have considered all use two kinds of equality: *definitional equality* and *book equality*. Definitional equality is handled in the frame by means of the reductions of the language. In general we expect the languages to have the property that definitional equality is decidable, whence it need not be expressed in the books. And there is never an assumption or a negation of definitional equality. On the other hand, book equality is introduced as a notion in the book, and particular book equalities can be proved, disproved or assumed in the book. In general there is no decision procedure for establishing book equality.

The effect of replacing impurities of the lambda calculus by book material is that one will have to live with the administration of many rather trivial book equalities. Let us use the term *strong book equalities* for them. Wherever we shift rules for definitional equality to the book we get such strong equalities, and, conversely, these strong equalities will be considered by others as suitable candidates for putting more definitional equalities into the frame.

As candidates for such strong book equalities we mention the equality axioms of sections 5.5 and 5.6.

It seems reasonable to give the strong equalities a separate status among the book equalities, in particular since much of the administrative work with these strong book equalities can be automated.

Right now this matter of strong book equality it is not more than a suggestion. The author has not acquired substantial experience with this, and such experience would be necessary for finding out whether it will be sufficiently efficient on the long run.

In order to describe some suggestions we denote ordinary book equality by IS, and strong book equality by ISSt (for types) and ISSo (for objects). Ordinary book equality is defined on any type X; if $x : X, y : X$ then $\mathrm{IS}(X, x, y)$ is a proof class. Strong book equality can also be expressed between types: If X : **type**, Y : **type**, then $\mathrm{ISSt}(X, Y)$ is a proof class. Moreover, we can express strong book equality between objects belonging to different but strongly book equal types. If X : **type**, Y : **type**, $u : \mathrm{ISSt}(X, Y)$, $x : X$, $y : Y$ then we may have $\mathrm{ISSo}^*(X, Y, u, x, y)$. A book axiom will express that $\mathrm{ISSo}(X, X, u, x, y)$ (where u refers to the reflexivity axiom $\mathrm{ISSt}(X, X)$) implies ordinary book equality $\mathrm{IS}(X, x, y)$. And there have to be axioms for strong equality for function values $F(x)$ and $G(y)$ if we have strong equality between F and G as well as between x and y.

We might also use strong equality for proof irrelevance. If P and Q are strongly book-equal proof classes, we may express the principle of irrelevance of proofs by postulating that whenever $p : P$ and $q : Q$ then the proofs p and q are strongly book-equal.

In [16] it was recommended to use the possibility to have two or more kinds of logic interwoven in one and the same book. In that case we have to introduce a different kind of proof classes for each kind of logic. Apart from assigning the type **prop** to a proof class, one might also handle **iprop** (for intuitionistic proofs) and **pprop** for positive (negation-free) proofs. It can be used like this: if p is some propsition then we can form Q with Q : **iprop**, and then $b : Q$ will mean that b is an intuitionistic proof for p. (In [16] it was recommended to use the word **prooftype** instead of the misleading word **prop**, and similarly **pprooftype** for negation-free logic.

In [16] this way of mixing various kinds of logic in one and the same book was recommended for treating constructions. The idea is that we want to be able to say that something is a construction, but we do not want to allow to say that something is *not* a construction. We do not want anybody to draw conclusions from the fact that it is not true that we have not constructed anything. Whether that might ever lead to trouble is doubtful, but it is just not the kind of thing we want to have expressed.

The matter with strong equality is somewhat parallel to this. If we have some kind of definitional equality in the frame, we are unable to express its negation in the books. So if we push that piece of definitional equality into the books, in the form of strong equality, we should not be able to handle its negation either. So it is reasonable to handle strong equality in terms of positive logic.

With the notation explained above this will be expressed by

$$ISSt(X, Y) : \mathbf{pprooftype}, \quad ISSo(X, Y, u, x, y) : \mathbf{pprooftype}.$$

6 The project $\Lambda\Sigma$

6.1 Introduction

It was expressed already in section 2 that at present $\Lambda\Sigma$ is to be considered as the name of a project rather than as a well-defined language with a satisfactory theory. The project is to define the upper right corner in the following diagram:

$$\begin{tikzcd}
\Delta\Lambda \arrow[r] & \Lambda\Sigma \\
\lambda \arrow[r] \arrow[u] & \lambda\sigma^* \arrow[u]
\end{tikzcd}$$

In the lower left corner we have λ, which is ordinary untyped lambda calculus. The one above it is the typed lambda calculus system $\Delta\Lambda$. In the lower right corner we have a modification $\lambda\sigma^*$ of the segment calculus $\lambda\sigma$ (see section 6.3). The problem is to find a definition and theory of a language $\Lambda\Sigma$ that has both $\Delta\Lambda$ and $\lambda\sigma^*$ as sublanguages.

The calculus $\lambda\sigma$ was introduced in [12] as an attempt to embed abbreviations for strings and telescopes into a system that takes them seriously. Usually one considers such things as a kind of sugaring, and treats them *ad hoc*, but this kind of abbreviations is bound to accumulate: abbreviations inside abbreviations, and then it is hard to build a dependable checker.

The matter is far from easy. One of the troubles with telescopes is that they contain variables to which there are references from other places, so we can get references to variables inside a telescope (or, more generally, a segment) that is only visible in the form of an abbreviation. Moreover, in different references to one and the same telescope we may have to have different sets for the names of the variables, in order to avoid name clashes. It is obviously a situation that asks for namefree calculus.

Some language theory, like the Church-Rosser property, was developed by H. Balsters in [1]. There has not been much activity in the subject since then.

6.2 The calculus $\lambda\sigma$

We shall not *define* $\lambda\sigma$ here, but just try to give an impression of what it is about.

We use the word *segment* for any string of applicators and abstractors in (untyped) lambda calculus. Avoiding namefree notation in this and the following examples, we mention the segment

$$(1) \qquad \langle\langle x\rangle[u]\langle x\rangle u\rangle\ [z]\ \langle\langle z\rangle[t]\langle\langle t\rangle x\rangle\rangle\ [v]$$

(note that the abstractors are untyped, so we have $[x]$ instead of $[x : X]$).

We want to make a calculus that treats segments in a way similar to terms, so we also want to have *segment variables*, and we want to interpret the elimination of segment abbreviations as a kind of β-reduction. As an example we take the abbreviation of the segment $\langle\langle x\rangle[u]\langle x\rangle u\rangle\ [z]$ by the single identifier η. In section 5.2 we showed how to do this for terms: If the abbreviation is $f := [u]\langle v\rangle u$ then elimination of that definition in a later occurreence of f is effectuated by a local β-reduction, carried out with the applicator-abstractor pair $\langle[u]\langle v\rangle u\rangle\ [f]$. So similarly we want to use for the case of the segment

$$\langle\langle\langle x\rangle[u]\langle x\rangle u\rangle\ [z]\rangle\ [\eta],$$

and there $[\eta]$ is an abstactor that introduces the segment variable η.

But lambda calculus is more than administration of abbreviations: it also concerns variables which are *not* going to be defined like the f above. Accordingly, we will also have undefined segment variables. And since segments may always contain variables that can be referred to, it becomes necessary to have some indication about the number of such referrable variables. To make it worse, some of the variables in the abstractors of a segment can again be segment variables, and the segments to be substituted for them can contain ordinary variables to which it is possible to refer even before these segment variables were replaced by the segments that were substituted for them.

In order to make that reference system more transparant, the suggestion was made in [12] to attach a kind of norm to the segment variables that shows the structure of the variables contained in them. That norm was called *frame*, but now that the word "frame" has got a completely different meaning it may be better to call that norm the *skeleton*.

The expressions that can be built by means of this segment calculus can sometimes be reduced (by β-reduction) to terms of the lambda calculus, but not always. The segments can begin to lead a life of their own, just like the variables in ordinary lambda achieve more than abbreviations.

We know that segments, in particular telescopes, can be used to describe complex abstract mathematical notions, like the notion "group". A group is usually considered as a mathematical object, but the notion "group" is not. A particular group is represented as a string, and the statement that it represents a group is expressed (with the terminology of [17]) that this string *fits* into the telescope "group". Extensive use of telescope abbreviations for treating modern mathematics was made by Zucker in [21].

This segment calculus is something like category theory. The notion "group" may refer to a particular category, represented by a telescope, but if we want to say "let C be any category", then we have to express that by "let η be any telescope", and there we need the telescope variable.

So we see that the title of [12] was not very fortunate: the scope of the paper is wider than the title suggests.

6.3 Extending to a typed calculus

What we have in mind with $\Lambda\Sigma$ is primarily a language in which the terms can be reduced to terms of a typed lambda calculus like $\Delta\Lambda$. That means that all segment variables can be eliminated by β-reductions.

If we omit all type information in such a calculus, we get the fragment $\lambda\sigma^*$ of $\lambda\sigma$, containing all $\lambda\sigma$-terms which are reducible to λ-terms. With this limited concept of $\lambda\sigma^*$ one might now say that the diagram of section 6.1 describes what we want. This vision on $\Lambda\Sigma$ will hopefully produce a system in which we can efficiently handle accumulated segment abbreviations, both for writing and for checking.

The matter may become harder if we want more than this, like handling segment variables as hinted at in the remark on category theory at the end of section 6.2.

References

[1] H. Balsters. Lambda calculus extended with segments. Ph.D. thesis, Eindhoven University of Technology 1986.

[2] H.P. Barendregt. Introduction to generalised type systems. Proceedings 3rd Italian Conference on Theoretical Computer Science., Eds. A. Bertoni a.o., World Scientific, Singapore 1989.

[3] N.G. de Bruijn. The mathematical language Automath, its usage, and some of its extensions. Symposium on Automatic Demonstration (Versailles, December 1968), Lecture Notes in Mathematics vol. 125, Springer Verlag 1970, pp. 29-61.

[4] ——. Automath, a language for mathematics. Séminaire Math. Sup. 1971. Les Presses de l'Université de Montréal 1973, 58 p.

[5] ——. AUT-SL, a single line version of Automath. Report, Department of Mathematics, Eindhoven University of Technology, 1971.

[6] ——. Some abbreviations in the input language for Automath. Report, Department of Mathematics, Eindhoven University of Technology, 1972.

[7] ——. A framework for the description of a number of members of the Automath family, Memorandum 1974-08, Department of Mathematics, Eindhoven University of Technology, 1974

[8] ——. Some extensions of Automath: the AUT-4 family. Report, Department of Mathematics, Eindhoven University of Technology, 1974.

[9] ——. Modifications of the 1968 version of Automath. Memorandum 1976-14, Department of Mathematics, Eindhoven University of Technology, 1976.

[10] ——. Some auxiliary operators in AUT-Π. Memorandum 1977-10, Department of Mathematics, Eindhoven University of Technology, 1976.

[11] ——. AUT-QE without type inclusion. Memorandum 1978-04, Department of Mathematics, Eindhoven University of Technology, 1978.

[12] ——. A namefree lambda calculus with facilities for internal definitions of expressions and segments. T.H. Report 78-WSK-03, Department of Mathematics, Eindhoven University of Technology, 1978.

[13] ——. A survey of the project Automath. In: To H.B. Curry: Essays in combinatory logic, lambda calculus and formalism, ed. J.P. Seldin and J.R. Hindley, Academic Press 1980, pp. 579-606.

[14] ——. Formalization of constructivity in Automath. In: Papers dedicated to J.J. Seidel, ed. P.J. de Doelder, J. de Graaf and J.H. van Lint. EUT-Report 84-WSK-03, ISSN 0167-9708, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1984, pp. 76-101.

[15] ——. Generalizing Automath by means of a lambda-typed lambda calculus. In: Mathematical Logic and Theoretical Computer Science, Lecture Notes in pure and applied mathematics, 106, (ed. D. W. Kueker, E.G.K. Lopez-Escobar, C.H. Smith) pp. 71-92. Marcel Dekker, New York 1987.

[16] ——. The use of justification systems for integrated semantics. In: Colog-88, P. Martin-Löf, G. Mints (editors). Lecture Notes in Computer Science vol 417, pp. 9-24, Springer Verlag, 1990.

[17] ——. Telescopic mappings in typed lambda calculus. To be published in Information and Computation.

[18] D.T. van Daalen. The language theory of Automath. Ph.D. Thesis, Eindhoven University of Technology, 1980.

[19] L.S. van Benthem Jutting. Checking Landau's "Grundlagen" in the Automath system. Ph.D. Thesis, Eindhoven University of Technology, 1977. Mathematical Centre Tracts Nr. 83, Amsterdam 1979.

[20] R.P. Nederpelt. Strong normalization in a typed lambda calculus with lambda structured types. Ph.D.thesis, Eindhoven University of Technology 1973.

[21] J. Zucker. Formalisation of classical mathematics in Automath. In: Colloques Internationaux du Centre National de la Recherche Scientifique, 249, (July 1975), pp. 135-145.

Finding Computational Content in Classical Proofs*

Robert L. Constable · Chet Murthy
rc@cs.cornell.edu · murthy@cs.cornell.edu
Department of Computer Science
Cornell University
Ithaca, NY 14853 USA

August 7, 1990

Abstract

We illustrate the effectiveness of proof transformations which bring out the computational content of classical proofs even in cases where it is not apparent. We state without proof a theorem that these transformations apply to proofs in a fragment of type theory and discuss the implementation of these in Nuprl. We end with a discussion of the revealing applications to Higman's lemma by the second author using the implemented system.

1 Introduction: Computational Content

informal practice

Sometimes we express computational ideas *directly* as when we say $2+2$ reduces to 4 or when we specify an algorithm for solving a problem; "use Euclid's GCD (greatest common divisor) algorithm to reduce this fraction". At other times we refer only indirectly to a method of computation as in Euclid's proof that there are infinitely many primes expressed in this form:

> For every natural number n there is a prime p greater than n. To prove this, notice first that every number m has a least prime factor; to find it, just try dividing it by $2, 3, \cdots, m$ and take the first divisor. In particular $n!+1$ has a least prime factor. Call it p. Clearly p cannot be any number between 2 and n since none of those divide $n!+1$ evenly. Therefore $p > n$. This ends the proof.

This proof implicitly provides an algorithm to find a prime greater than n. Namely, divide $n!+1$ by all numbers from 2 to itself. We can "see" this algorithm in the proof, and we say that the proof has *computational content*.

In day-to-day mathematics people use a variety of schemes to keep track of computational content. There is no single way or systematic way. People use phrases like "reduce b to b'" or "apply algorithm f." They also tend to use phrases like "this proof is constructive" or "the proof shows us how to compute." It is also sensible to say "we treat all of the concepts constructively." To understand the last phrasing notice that it is not possible to tell from the usual statement of

*This research was supported in part by NSF grant CCR-8616552 and ONR grant N00014-88-K-0409

a theorem alone whether its proofs must be constructive (or whether any proof is). To see this consider the statement:

> There are two irrational numbers, a and b such that a^b is rational.
>
> Here is a nonconstructive proof of it. Consider $\sqrt{2}^{\sqrt{2}}$; it is either rational or not. If it is, then take $a = \sqrt{2}$ and $b = \sqrt{2}$. Otherwise, take $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$, then $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$.

This proof does not allow us to compute the first 10 digits of a since there is no way, *given in the proof*, to tell whether $a = \sqrt{2}$ or $a = \sqrt{2}^{\sqrt{2}}$. The statement of the theorem does not tell whether a constructive proof is required, in contrast to this form which does:

> We can exhibit two algorithms A and B for computing irrational numbers, a and b respectively, such that a^b is rational. Here is a proof which suffices for both theorems. Take $a = \sqrt{2}$ and $b = 2 \cdot \log_2(3)$. Notice that there are algorithms A and B to compute these numbers.

Often when we say "there exists an object b having property P" we really mean that we can explicitly exhibit an algorithm for computing b. This use of language is so common that to convey computational ideas we can just say "there exists" is meant constructively. Another example of how language is used to convey computational intention is this. Let us show:

> At least one of $(e+\pi)$ or $(e*\pi)$ is transcendental. Suppose not, then both are algebraic. So the coefficients of $(x-\pi)(x-e)$, i.e. of $x^2 - (e+\pi) * x + (e * \pi)$, are algebraic. But then according to the theorem that the field of all algebraic numbers is algebraically complete, all the roots are algebraic. But this contradicts the fact that e and π are both transcendental.

The use of "or" in this case does not tell us that we can decide which disjunct is true. But when we say every number n is even or odd, we notice that we can decide which. It is also common to say, "for all x $P(x)$ or $Q(x)$," when we mean that we can decide for any x whether it is $P(x)$ or $Q(x)$ that is true. Again sometimes a proof will tell us that, as in the case of a proof:

> For all natural numbers n, n is prime or n is composite. We can prove this by testing whether n is prime.

That proof allows us to decide the *or*. So another way to convey computational content is to indicate that *or* is to be interpreted in a constructive sense.

systematic accounting

There have been various attempts to keep track of computational ideas systematically. The constructive or Intuitionistic use of language is one of the oldest and most throughly studied. Another way is to explicitly use algorithms to define functions and limit assertions to equalities between them as in Skolem's account of arithmetic [19]. There are formal languages in which programs are explicitly used to express algorithms. Another way is to introduce a double set of concepts, say function and computable function, classical and constructive logical operators, and so forth. This leads to a highly redundant language, so there have been efforts to reduce the duplication. These result in certain epistemic logics [17, 11].

The approach of using an Intuitionistic or constructive core language has been extensively studied by philosophers, logicians, mathematicians and computer scientists for most of this century. Many deep and beautiful results about this method are now known, and they are being applied extensively in computer science (already since the late 60's). The applications are to the design of programming languages [16, 2], to programming methodology [12, 3], to program verification and formal methods [4, 3], and to automated reasoning [15]. This paper discusses further applications, in particular to foundational issues about the nature of computation and to problems in the semantics of functional programming languages. We believe that this "constructive language" approach to accounting for computational content has so far been the most direct approach, although for historical reasons going back to the highly emotional debates between Brouwer and Hilbert the approach seems a bit "controversial". Moreover, there are strong connections to issues in the philosophy of mathematics which are still unresolved and some people are loath to appear to be taking a stand on issues about which they have little concern or knowledge. It is fair to say that many scientists have used these concepts and results without regard for the philosophical issues. This paper takes no philosophical stand. We try to continue the tradition of using the techniques of constructive logic to shed light on fundamental *scientific issues.*

constructive language

A natural starting place to understand the systematic use of constructive language is with the notion of *existence.* As far back as the ancient Greeks the notion of existence in mathematics was identified with the concept of a construction (by ruler or compass). The constructive interpretation of the quantifier $\exists x \in T.P$, meaning that there is an object of type T such that property P holds of it, is that we can construct the object in T, call it t, and know that P is true when x refers to t, i.e. the formula with some description of t substituted for x in P is true. The computational interpretation here is clear if we think of a description of t as an algorithm for building the object. Under this interpretation, a key step in understanding constructive language is knowing how to describe the objects in a type. Before we can be specific about existential statements, we must make some commitments about what ways we allow for describing objects. Looking to simple examples like the natural numbers for guidence, we see that there are irreducible descriptions, such as 0, and compound ones, such as $0+0$. Following Martin-Löf [12] we adopt the notion that our language of descriptions will be based on the idea that some are *canonical*, such as 0, and some are not, such as $0+0$. The key principle about the noncanonical description is that they can be *computed* to canonical ones. This is a precise version of the notion that a description tells us how to build the object because the canonical name is a direct description of the object. These ideas can be made precise in Martin-Löf's type theories.

Once we have settled on a computational interpretation of existence, we then want to make sure that the other quantifiers and logical operations preserve it. This forces us to interpret P or Q as meaning that we know which of P or Q is true since we can build a noncanonical description os an object based on a case analysis, e.g. take the statements:

$$\sqrt{2}^{\sqrt{2}} \text{ is rational or not } \sqrt{2}^{\sqrt{2}} \text{ is rational,}$$

$$\text{if } \sqrt{2}^{\sqrt{2}} \text{ is rational take } a = \sqrt{2} \text{ else take } a = \sqrt{2}^{\sqrt{2}} \ .$$

Another important feature we would like in our computational understanding of language is that when we assert

for all x of type S there is a y of type T such that P

(symbolized as $\forall x \in S.\exists y \in T.P$) we mean that there is an algorithm to compute a function from S to T which produces the witness object in T given any object s of S. This suggests that we interpret the universal quantifier $\forall x \in S.P$ as asserting that there is an algorithm taking any object of type S and producing a proof of P.

The simplest statements are built out of terms (the statements themselves can be construed as terms). There is no computational content to a meaningful irreducible, such as 0 or a. But for compound terms, such as $2+2$ or $3+1$, we might say that the computational content is the set of all computations or reductions of the term. So the exact meaning depends on how the term is built, in the examples, how $+$ is defined.

The atomic statements we consider have the form $P(a_1; \cdots; a_n)$ for a_i terms, P an operator. Associated with this term is a computation or set of computations. We take these to be the computational content. For example, the computation associated with the statement $4 = 2+2$ is a justification from the axioms of $+$. Compound statements are formed from $\forall, \exists, \vee, \wedge, \Rightarrow$ with their computations being naturally defined in terms of the computations of the relevant subsidiary statements. Thus, the computations of $a = b \Rightarrow b = a$ are the computations which transform computations of $a = b$ into computations of $b = a$. The computations of $\exists x \in T.P$ are those which compute a t in T and a computation of P where t has been substituted for x.

seeing computational content

One advantage of constructive language is that it allows us to see computational content directly; we can read it off from a proof of a theorem stated constructively. In fact, constructive proofs are so transparent that we can just see the content and extract it automatically. Here is an example. Consider the theorem

Theorem 1 *root:* $\forall x \in \mathsf{N}.\exists r \in \mathsf{N}.r^2 \leq n \leq (r+1)^2$
Proof. *: Let n be given in* N*, by* induction *on n show* $\exists r.\ r^2 \leq m \& n < (r+1)^2$.

Base: Choose $r = 0$, use simple arithmetic.

Ind: Assume $h: \ \exists r.\ r^2 \leq n \ \&\ n < (r+1)^2$ using hyp h, choose r_0
$d: (r_0+1)^2 \leq (n+1) \vee n+1 \leq (r_0+1)^2$ by arithmetic.
Show $\exists r.\ r^2 \leq (n+1)\ \&\ (n+1) < (r_0+1)^2$ by *or elimination* on d
If $(r_0+1)^2 \leq (n+1)$ then $r = (r_0+1)$, use arithmetic
If $(n+1) < (r_0+1)^2$ then $r = r_0$, use arithmetic
QED

The computational content of this proof is given by the function:

$$
\begin{array}{l}
\lambda n.\ ind(n;\ <0,\ arith>; \\
\quad u.\lambda h.\ spread(h:\ r_0, isroot. \\
\qquad seq(arith : disjunct. \\
\qquad\quad decide(disjunct; \\
\qquad\qquad lesseq.\langle r_0+1, arith\rangle; \\
\qquad\qquad qtr\ .\langle r_0, arith\rangle))))
\end{array}
$$

This can be extracted automatically from the proof. It is an algorithm which computes an integer square root and a proof that the number computed is a root.

We can see computational content in classical proofs as well. Consider the following nonconstructive proof that a root exists (nonconstructive because it proceeds by contradiction).

Theorem 2 $\forall n \in \mathsf{N}.\exists r \in \mathsf{N}.r^2 \leq n\ \&\ n < (r+1)^2$

Proof. ("Classical"):
Given any $n \in \mathsf{N}$,
Assume $\forall r \in \mathsf{N}.r^2 > n \vee (r+1)^2 \leq n$

Let $G = \{r : \mathsf{N} | r^2 > n\}$
$S = \{r : \mathsf{N} | (r+1)^2 \leq n\}$

Note 1. $S \cap G =$ void
2. S's elements are smaller than G's

Let $r_0 > n$ be the least element of G
$*$ $r_0^2 > n$ by definition of G
$**$ $r_0 - 1 \in S$ since r_0 is least
$(r_0 - 1 + 1)^2 \leq n$ by def of S
$r_0^2 \leq n$
QED

Is there a way in which we can extract computational content from these classical proofs in which it is so transparent? The answer is yes, and indeed much more can be done. G. Kriesel [9, 18] discovered a remarkable theorem about this based on results of Gödel. Then in 1978 H. Friedman [8] drastically simplified the method. We will examine these techniques in the next section.

2 Extracting Computations by Translation : The A-Translation

The A-translation is a short name for the method that H. Friedman [8] used to simplify and extend G. Kreisel's theorem that any classical proof of $\exists x \in \mathsf{N}.P$ where P is decidable can be transformed into a constructive proof of the same statement. This result is itself effective, and an implementation of it would enable us to extract content from certain classical proofs, such as Theorem 2, by transforming it in this way and then extracting from the constructive proof. The usual presentation of this method involves two steps, first translating the classical language into a constructive one, then massaging the transation so as to bring out the content. The first step is due to Gödel.

Kolmogorov and Gödel translations

In 1932 Gödel showed how to interpret classical language into constructive. The key ideas are to notice that one way to understand the classical meaning of *or* is to say that P *or* Q means that we cannot imagine any other possibility; that is we cannot say that both $\neg(P)$ and $\neg(Q)$. So P *or* Q could be read as $\neg(\neg(P) \wedge \neg(Q))$. The key to understanding the classical use of the existential quantifier is to say that $\exists x \in T.P$ means it cannot be the case that for all x in T, $\neg(P)$ holds. If we use the symbols $\circledvee$ and $\circled{\exists}$ for the classical disjunction and existential quantifier respectively, then the translations are:

$$\begin{array}{rcl} P \circledvee Q & \equiv & \neg((\neg(P) \wedge \neg(Q))) \\ \circled{\exists} x \in T.P & \equiv & \neg(\forall x \in T.\neg(P)). \end{array}$$

Classically these are laws relating the operations mentioned, but constructively they are definitions of new operators. Applying these definitions to a classical statement results in a constructive one which explains the classical statement in computational terms. Gödel defined a complete embedding of classical arithmetic as follows:

Gödel Transformation

$(A \& B)^\circ \ = \ A^\circ \& B^\circ$
$(A \Rightarrow B)^\circ \ = \ A^\circ \Rightarrow B^\circ$
$\forall x.B^\circ \ = \ \forall x.\ B^\circ$
$(A \circledvee B)^\circ \ = \ \neg(\neg A^\circ \ \& \ \neg B^\circ)$
$(\circled{\exists} x.B)^\circ \ = \ \neg\forall x.\ \neg B^\circ$
$atomic^\circ \ = \ atomic \qquad \perp^\circ \ = \ \perp$
Note: $\neg B \ = \ (B \Rightarrow \perp)$
where $\perp$ denotes the false proposition

Example: $(P \circledvee \neg(P))^\circ \ = \ \neg(\neg(P) \ \& \ \neg\neg(P))$, for P atomic

Theorem 3 (Gödel) *If F is provable in classical arithmetic from Γ, written $\Gamma \vdash_{PA} F$, then F° is provable in constructive arithmetic from Γ°, written $\Gamma^\circ \vdash_{HA} F^\circ$.*

Friedman's A-translation

In 1978 Friedman [8] gave a new syntactic proof that one could automatically transform classical proofs of Σ_1^0 sentences (existential sentences where the body is decidable) into constructive proofs. This proof was in two steps, the first consisting of the just-defined double-negation translation. The second step can be defined in numerous ways, depending upon the exact kind of source and destination logics we wish to consider. We will define it as Friedman did, and then show a simplification, originally due to Leivant [10], which makes the task of A-translation essentially effortless.

The purpose of A-translation is to transform a Gödel-translated proof of a Σ_1^0 sentence back into a proof of the original sentence.

Definition 2.1 (A-Translation) *The A-translation of a proposition ϕ is accomplished by simultaneously disjoining every atomic formula of ϕ with the proposition A, and is written ϕ^A.*

For example, $(a = b \wedge b = c)^A \equiv (a = b \vee A) \wedge (b = c \vee A)$. We can prove the following theorem:

Theorem 4 (Friedman) *If $\Gamma \vdash_{HA} F$ then $\Gamma^A \vdash_{HA} F^A$ for any A.*

Consider now a classical proof

$$x : \mathsf{N} \vdash_{PA} \exists y \in \mathsf{N}.\Phi(x, y),$$

and let $A = \exists y \in \mathsf{N}.\Phi(x, y)$. The Godel-translation theorem lets us construct

$$x : \mathsf{N} \vdash_{HA} \neg\neg(\exists y \in \mathsf{N}.\Phi(x, y))$$

from which, by A-translation, we get:

$$x : \mathsf{N} \vdash_{HA} ((\exists y \in \mathsf{N}.\Phi(x, y)^A) \Rightarrow A) \Rightarrow A,$$

where, as we said, A is the original goal. Now, when Φ is decidable, it is simple to show

$$x : \mathsf{N} \vdash_{HA} (\exists y \in \mathsf{N}.\Phi(x, y)^A) \Rightarrow A,$$

and putting together the last two proofs (we refer to this as "Friedman's top-level trick"), we get

$$x : \mathsf{N} \vdash_{HA} \exists y \in \mathsf{N}.\Phi(x, y).$$

While this passage from the double-negated proof to the A-translated, double-negated proof looks quite complicated, what's really going on is that we first translate the double-negated proof from a constructive logic into a minimal logic, and then replace every instance of $\perp$ with A. Thus, if we were already in a minimal logic, we would not have to do the first step, and we could just replace every $\perp$ with A.

In any case, regardless of what version of the A-translation we use, we can prove

Theorem 5 (Friedman) *Given a classical proof*

$$x : \mathsf{N} \vdash_{PA} \exists y \in \mathsf{N}.\Phi(x, y),$$

where Φ is decidable, we can construct a constructive proof of the same,

$$x : \mathsf{N} \vdash_{HA} \exists y \in \mathsf{N}.\Phi(x, y),$$

via double-negation translation, followed by A-translation, followed by the top-level trick.

In the following development of the translation of a particular simple argument, we will assume that the target logic of the double-negation translation is minimal. Hence, the A-translation simply replaces $\perp$ in the double-negated proof with A. Since the logic will be minimal, this replacement operation does not perturb the proof, and so the double-negated proof is still a valid proof of the A-translated sentence.

Let's say that again. If we assume that double-negation translation leaves us in a minimal logic, then a proof of a double-negated sentence is *also* a proof of any A-translation of that sentence. This means that A-translation is really just a "name change", which does nothing of real importance, whereas double-negation translation does all the real work of extracting computations. So we will focus upon the effect of double-negation translation, to see how it extracts computational content.

3 Hidden Constructions

One advantage of the A-translation method is that it can uncover constructions which are difficult to see. This idea can be illustrated with a very simple example, but it has also been employed by C. Murthy [13] to obtain a computation (albeit a grossly infeasible one) from a classical proof of Higman's Lemma. The simple example is enough to illustrate several important concepts.

Let $\mathsf{N} \equiv \{0, 1, 2, \cdots\}$. Consider a function $f : \mathsf{N} \to \mathsf{N}$ whose values are all either 0 or 1. Call such a function *binary*. Here is a trivial fact about them whose proof we will call **pf1**.

> For all $f : \mathsf{N} \to \mathsf{N}$ which are *binary*, there exist $i, j \in \mathsf{N}$, $i < j$ such that $f(i) = f(j)$. Here is a classical proof. Either there are infinitely many 0's in the range of f (abbreviated $Zeros(f)$) or not. If not, then there are infinitely many 1's ($Ones(f)$). In the first case, choose i, j to be two distinct numbers on which f is 0. In the other case, choose i, j to be distinct numbers on which f is 1.

This is a highly nonconstructive proof. Indeed it is quite difficult to see any construction in it at all. Whereas there are quite trivial constructive proofs such as the following which we call **pf2**.

> Consider $f(0), f(1), f(2)$. If $f(0) = f(1)$ then $i = 0, j = 1$. If $f(0) \neq f(1)$, then if $f(0) = f(2)$ take $i = 0, j = 2$. If $f(0) \neq f(2)$, then it must be that $f(1) = f(2)$ since f is *binary*, so take $i = 1, j = 2$.

This proof is very constructive, and we see that the extracted algorithm is just

```
if f(0) = f(1)
then i := 0,  j := 1
else if f(0) = f(2)
    then i := 0,  j := 2
    else i := 1,  j := 2
```

It is interesting that the A-translation of **pf1** does not produce the same algorithm as that obtained from **pf2**. We will see the difference below.

We will now examine some of the key steps in a more rigorous treatment of the first proof, **pf1**. We will try to outline just enough of the argument that the hidden construction becomes visible. First some definitions. We write $Ones(f)$ to mean that there are infinitely many 1's in the range of f, similarly for $Zeros(f)$. The classical definitions are:

$$\begin{aligned} Ones(f) &\equiv \forall i \in \mathsf{N}. \circledexists j \in \mathsf{N}. i < j \wedge f(j) = 1 \\ Zeros(f) &\equiv \forall i \in \mathsf{N}. \circledexists j \in \mathsf{N}. i < j \wedge f(j) = 0. \end{aligned}$$

A key lemma is that there are either infinitely many ones or infinitely many zeros. It is expressed as:

$$\forall f \in \mathsf{N} \to \mathsf{N}. binary(f) \Rightarrow Ones(f) \circledvee Zeros(f).$$

When Godel-translated this is

$$\forall f \in \mathsf{N} \to \mathsf{N}. binary(f) \Rightarrow \neg(\neg(Ones(f)) \wedge \neg(Zeros(f)))$$

which is

$$\forall f \in \mathsf{N} \to \mathsf{N}.binary(f) \Rightarrow ((Ones(f) \Rightarrow \bot) \wedge (Zeros(f) \Rightarrow \bot)) \Rightarrow \bot.$$

The informal classical proof is from the classical fact that either $Ones(f)$ or $\neg(Ones(f))$. If $\neg(Ones(f))$, then there is some point x_0 such that for $y > x_0$, $f(y) = 0$. This means that $Zeros(f)$. Let us look at the proof more carefully. We will give a constructive proof of the Godel-translated sentence, **L0**:

Show:$\forall f \in \mathsf{N} \to \mathsf{N}.binary(f) \Rightarrow ((Ones(f) \Rightarrow \bot) \wedge (Zeros(f) \Rightarrow \bot)) \Rightarrow \bot$
Proof. assume $f : \mathsf{N} \to \mathsf{N}$, $binary(f)$,
$Ones(f) \Rightarrow \bot$, $Zeros(f) \Rightarrow \bot$
"there is an x_0 beyond which no values are 1's"
This corresponds to trying elimination on $Ones(f) \Rightarrow \bot$ to get $\bot$.
Show $\forall x \in \mathsf{N}.\neg\neg(\exists y \in \mathsf{N}.x < y \wedge f(y) = 1)$
Proof. arb $x_0 : \mathsf{N}$, show $(\exists y \in \mathsf{N}.(x < y \wedge f(y) = 1) \Rightarrow \bot) \Rightarrow \bot$
$a1$:assume $\exists y \in \mathsf{N}.(x < y \wedge f(y) = 1) \Rightarrow \bot$
(this is equivalent to $\forall y \in (x < y \Rightarrow f(y) \neq 1)$., beyond x_0 all values are 0)
plan to get $\bot$ by elim on $Zeros(f) \Rightarrow \bot$
show $Zeros(f)$, i.e. $\forall z \in \mathsf{N}.\neg\neg(\exists y \in \mathsf{N}.z < y \wedge f(y) = 0)$
Proof. arb $z : \mathsf{N}$ sho $(\exists y \in \mathsf{N}.(z < y \wedge f(y) = 0) \Rightarrow \bot) \Rightarrow \bot$
a2: assume $\exists y \in \mathsf{N}.(z < y \wedge f(y) = 0) \to \bot$
(now know from a1, a2 that can't have f be either 0 or 1, get $\bot$)
let $w + max(z, x_0) + 1$
$d : f(w) = 0 \ \vee \ f(w) = 1$ by binary (f)
show $\bot$ by cases on d above
if $f(w) = 0$ then $z < w \ \& \ f(w) = 0$ so
$\exists y \in \mathsf{N}.(z < y \wedge \ \& \ f(y) = 0)$
$\bot$ by elim on a2

if $f(w) = 1$ then $x_0 < w \ \& \ f(w) = 1$ so
$\exists y \in \mathsf{N}.(x < y \wedge \ \& \ f(y) = 1)$
$\bot$ by elim on a1

$\bot$
Qed so zeros(f)
$\bot$ by elim on $\neg(Zeros(f))$
Qed Ones (f)
$\bot$ by elim on $\neg(Ones(f))$
QED

The other two key steps are to show that $Ones(f)$ and $Zeros(f)$ each imply the goal.

The classical goal $\exists F \equiv \exists i, j \in \mathsf{N}.i < j \wedge f(i) = f(j)$, and the double-negation translation of this is $G \equiv \neg\neg(\exists F)$. We must prove the two lemmas:

$$\begin{aligned} \mathbf{L1} &: Ones(f) \Rightarrow G \text{ and} \\ \mathbf{L2} &: Zeros(f) \Rightarrow G. \end{aligned}$$

Let us examine a proof of **L2**.

Show: $Zeros(f) \Rightarrow G$
a1: assume $\forall x(\exists y(x < y \ \& \ f(y) = 0)) \rightarrow \perp) \rightarrow \perp$
show $(\exists i, j \ F \rightarrow \perp) \rightarrow \perp$
a2: assume $(\exists i, j \ . \ F \rightarrow \perp)$
could try to show $\exists i, j \ . \ F$ and use $\rightarrow$ elim, but that is too direct, we need to use $Zeros(f)$
a3: $(\exists y(0 < y \ \& \ f(y) = 0) \rightarrow \perp) \rightarrow$ by allel a1, on 0
show $\exists y(0 < y \ \& \ f(y) = 0)\perp$ for $\rightarrow$ elim
assume $\exists y.(0 < y \ \& \ f(y) = 0$ to show $\perp$
choose y_0 where $0 < y_0 \ \& \ f(y_0) = 0$
a4: $(\exists y(y_0 < y \ \& \ f(y) = 0)) \rightarrow \perp) \rightarrow \perp$ by allel a1 on y_0
show $\exists y(y_0 < y \ \& \ f(y) = 0) \rightarrow \perp$ for $\rightarrow$ elim
a5: assume $\exists y(y_0 < y \cdots)$
choose y_1 where $y_0 < y_1 \ \& \ f(y_1) = 0$.

$\exists F$ by $\exists$ intro y_0, y_1
$\perp$ by elim a2
$\perp$
$\perp$
$\perp$
QED

Let's put all of this together:
Theorem: $\forall f \in \mathsf{N} \rightarrow \mathsf{N}.binary(f) \Rightarrow \neg\neg(\exists F)$:

arb $f : \mathsf{N} \rightarrow \mathsf{N}$, $b : binary(f)$, show $\neg\neg(\exists F)$
a1: assume $\neg(\exists F)$, show $\perp$
seq $a2$:$\neg(\neg(Zeros(f))\&\neg(Ones(f)))$
by lemma **L0**, with parameters f, b
$\perp$ by *function elimination* on $a2$
show $\neg(Zeros(f))\&\neg(Ones(f))$ by *and introduction*
show $\neg(Zeros(f))$
a3: assume $Zeros(f)$, show $\perp$
seq h:$\neg\neg(\exists F)$ by lemma **L1** with parameter $a3$
$\perp$ by elim h on $a1$
$\neg(Zeros(f))$
show $\neg(Ones(f))$
a4: assume $Ones(f)$, show $\perp$
seq h:$\neg\neg(\exists F)$ by lemma **L2** with parameter $a4$
$\perp$ by elim h on $a1$
$\neg(Ones(f))$
$\neg(Zeros(f))\&\neg(Ones(f))$
$\perp$
QED

As we said before, we assume that this proof of $\neg\neg(\exists F)$ is in a minimal logic; hence, we can automatically generate, for any A, a proof of

$$x : \mathsf{N}, binary(f) \vdash_{HA} (\exists F \Rightarrow A) \Rightarrow A,$$

and from this, by Friedman's top-level trick, a proof of

$$x : \mathsf{N}, binary(f) \vdash_{HA} \exists i, j \in \mathsf{N}. i < j \wedge f(i) = f(j).$$

4 Mechanizing the Translations in Nuprl

The work thus far presented in this paper came out of a two-year long project to understand the meaning of Friedman's translations. As part of this project, we implemented Friedman's metatheorem as a proof transformation procedure which translated proofs in a classical variant of Nuprl into proofs in constructive Nuprl. Unfortunately, we cannot implement Friedman's metatheorem for all of Nuprl; that is, the *entire* Nuprl type theory, with the addition of the axiom of excluded middle, is not a suitable candidate for Friedman's translation. However, a carefully chosen subtheory is a suitable candidate, and we will spell out restrictions on Nuprl which yield a theory Nuprl°, which *can* be translated.

Having spelled out the subtheory of Nuprl which, when made classical, is translatable, we will briefly discuss the mechanized implementation of the "binary" theorem, and then discuss parts of the translated, extracted computation.

Finally, we will describe some results that came out of this project. They are described in complete detail in the second author's thesis [13], and we will only state them here.

4.1 Effective Translation

The problem in translating a constructive type theory such as Nuprl is made difficult by the fact that Nuprl is not really a constructive *logic*. We distinguish between a type theory and a logic by focusing upon the linguistic use of propositions and their proofs (inhabitants/members). In a logic,we may not treat a proposition as a type, i.e. having members, but only as being proven. Thus, in a logic, we may not quantify over propositions, or use their proofs (inhabitants) by name. Rather, we can only use the fact of the inhabitation of the proposition. In a type theory, on the other hand, we are free to use inhabitants of propositions by name. This is exactly where our problems in translation come up, because when we prove a sentence like $\exists x \in T.P$ classically, the proof is not a means of constructing a member t of T, such that P holds of that member. Rather, the proof is simply a guarantee that we can never construct a counter-example to the existence of such a t. Thus, we should not be able to reason from the *value* of the inhabitant; rather, we should only be able to reason from the *fact that* the proposition is inhabited. This conforms well with the classical notion that a proposition is simply proven; the proofs of a proposition are indistinguishable, since they are noncomputational. Likewise, we should always be able to reason that if $P \Leftrightarrow Q$, then $\Phi(P) \Leftrightarrow \Phi(Q)$, where P, Q are propositions, and Φ is a predicate on propositions. This means that propositional formation operators must bi-implicatively respect bi-implication.

These ideas can be made precise by defining a theory much like Nuprl, called Nuprl°, such that proofs in Classical Nuprl° can always be translated back into Nuprl°. Then, we define a trivial mapping from Nuprl° to Nuprl (an erasing of certain annotations) which finishes the job. There are three basic restrictions we make upon Nuprl proofs to make them into Nuprl° proofs:

- Every proposition in every sequent must be statically well-formed. That is, the well-formedness of a proposition should not depend upon the truth (inhabitation) of some other proposition. Some examples of this problem come up in uses of the "set" type, which is used in Nuprl to hide computational content.

- Every proposition-forming operator should respect $\Leftrightarrow$, as explained before. This restriction is simple to enforce, and it comes about because we wish to enforce the "logical" nature of our language.

- the "proposition-hood" of a term must be syntactically decidable; that is, we must be able to statically decide whether a given term in a given sequent is a proposition or not, and no term which is a proposition in one sequent can be a data-value, or data-type, in another sequent (e.g. a parent sequent or subsidiary sequent). We enforce this condition by annotating every term (and every subterm) with either a P (for proposition) or a D (for data), and verifying that certain compatibility conditions hold between adjacent sequents in proof trees.

With these three conditions, we can show that, for a suitable subset of the type constructors of the Nuprl type theory, e.g. Π-types, Σ-types, universes, higher-order predicates and data, integers, lists, etc, the Classical Nuprl$^\circ$ theory, with excluded middle, is translatable automatically into constructive Nuprl$^\circ$. We can then embed Nuprl$^\circ$ back into Nuprl by simply erasing the annotations. Hence, we have a set of sufficient conditions for the success of the translation in Nuprl.

One wonders if these restrictions are really relevant; if these restrictions actually exclude pathological cases of proofs which are intrinsically not translatable, and include important cases of proofs which we must translate. The second part of this question is easily answered: our translation effort, which succeeded in translating Higman's Lemma, attests to the tractability of doing mathematics within this fragment of Nuprl. To answer the first part of the question, we showed that, for a particular formalization of the axiom of choice which is trivially provable in Nuprl, its double-negation translation is not provable in Nuprl. It can be written thus:

$$\forall x \in Dom. \exists y \in Rng. \Phi(x, y) \Rightarrow \exists f \in Dom \to Rng. \forall x \in Dom. \Phi(x, f(x)),$$

and its proof in Nuprl is $\lambda\ h.\langle \lambda\ x.h(x).1,\ \lambda\ x.h(x).2\rangle$. We can think of this as a function which, given an input function which stands as a proof of $\forall x \in Dom. \exists y \in Rng. \Phi(x, y)$, *splits* the input function into the two parts, the first of which computes values in Rng, and the second part of which witnesses the correctness of the first part.

But this proof assigns a name, h, to the proof of $\forall x \in Dom. \exists y \in Rng. \Phi(x, y)$, which we expressly forbade in our conditions above. For this $\forall\exists$ sentence is an object of proof, hence a proposition, and to assign a name to the proofs (inhabitants) of a proposition is to quantify over it, which, again, is forbidden in our fragment of Nuprl. The double-negation translation of this axiom is equivalent to the following sentence (where Φ is double-negated):

$$\forall x \in Dom. \neg\neg(\exists y \in Rng. \Phi(x, y)) \Rightarrow \neg\neg(\forall x \in Dom. \exists y \in Rng. \Phi(x, y)),$$

and we showed that this sentence is not true in the standard model of Nuprl [1]. Thus, we cannot use the axiom of choice to construct functions by induction in our classical proofs. Instead, we must formalize functions as binary relations for which certain existence and uniqueness predicates hold.

Using such a formulation, we can then do almost all of the reasoning we wish to about functions, paying only a small price in terms of cumbersomeness of the logic.

We have not yet discussed the A-translation, nor have we discussed the actual implementation of our proof translations. However, the actual implementation details follow rather straightforwardly from the restrictions on the forms of proofs, and the A-translation follows likewise. The difficult part is restricting the Nuprl logic to make translation possible at all. In the end, though, we prove

Theorem 6 (Conservative Extension for Nuprl°) *Given a proof in Classical Nuprl° of a* Π^0_2 *statement, we can construct a proof in (constructive) Nuprl° of the same statement.*

4.2 A Formalization of the Binary Theorem

We formalized the binary theorem in Nuprl, giving it a classical proof essentially identical to the proof described earlier in this paper. The sentence actually proven was:

$$\forall f \in \mathsf{N} \to \{0,1\}.\exists i, j \in \mathsf{N}.i < j \wedge f(i) = f(j).$$

As described before, we proved $Ones(f) \otimes Zeros(f)$, and from this we concluded the actual goal. We then employed an automatic double-negation/A-translation tactic which converted the proof to a constructive one, from which we extracted the computational content, which we then executed. This program yielded the same numeric answers as the program derived earlier. Later, we employed a more intelligent translation scheme, and we list below the computation which was extracted from this scheme, after normalization and trivial compression:

```
\f.int_eq(f(1);0;
      int_eq(f(2);0;
                 <1,<2,axiom>>;
                 int_eq(f(3);0;
                        <1,<3,axiom>>;
                        <2,<3,axiom>>));
          int_eq(f(2);0;
                 int_eq(f(3);0;
                        <2,<3,axiom>>;
                        int_eq(f(4);0;
                               <2,<4,axiom>>;
                               <3,<4,axiom>>));
                 <1,<2,axiom>>))
```

Written in pseudo-code, this program is

```
if f(1)=0 then
  if f(2)=0 then
    1,2
  else if f(3)=0 then
    1,3
  else
```

```
    2,3
else if f(2)=0 then
  if f(3)=0 then
    2,3
  else if f(4)=0 then
    2,4
  else
    3,4
else
  1,2
```

Note that, as we said before, this strategy requires the first *four* terms of f.

We formalized Higman's Lemma in Nuprl, and translated it using the same apparatus we developed for the binary theorem, extracting constructive content in the same way. This program was approximately 12 megabytes in size, which made it infeasible to run the program on any nontrivial inputs. Later, we formalized another classical argument, in the same vein as Higman's Lemma, and again we extracted a program of 19 megabytes in size. We were dismayed by the sizes of the extracted programs, but, as we shall discuss in the next section, this was purely a matter of the exact translation implementation (which was quite inefficient), and not an intrinsic problem with double-negation/A-translation.

5 The Algorithmic Content of Classical Proofs

Through our work in trying to understand the meaning of these translations we discovered some startling new facts about the connections between classical reasoning and computation. We can summarize them in the following two theorems:

Theorem 7 *Friedman's method (double-negation/A-translation) is exactly a continuation-passing-style [7, 14] compilation upon the "classical witness" from a proof in a classical theory (say, Peano Arithmetic).*

We have learned that many of the different double-negation translations can be understood in terms of their effect upon programs, and not just in terms of their effect upon proofs. To wit, we have shown that several double-negation translations in fact fix the order of evaluation of expressions in a functional language, thus providing an explanation of various eager and lazy computation schemes in terms of each other, and hence in a single functional language.

Theorem 8
- *A particular modified Kolmogorov double-negation translation fixes a call-by-name evaluation order on functional program expressions.*
- *A modified Kuroda translation fixes an eager (call-by-value) evaluation order on functional programs.*
- *a further modification of the Kolmogorov translation makes pairing an eager operation, and allows either by-value or by-name lambda-application.*

These discoveries tell us that classical proofs have implicit algorithmic content, and that this algorithmic content is made explicit by the double-negation/A-translations. They tell us that the proof-translation method of double-negation/A-translation is identical to the program translation method of continuation-passing-style (CPS) translation, and that classical proofs can be interpreted as nonfunctional programs (enriched with a standard nonlocal control operator called "control" ($\mathcal{C}$), a relative of call-with-current-continuation (call/cc)) [5, 6]. They provide a rational, *algorithmic* foundation for Friedman's translation method, based on its effect on computations, and upon programs; that is, in a manner which is not characterized by effects upon proofs and sequents, but rather on a "semantic" basis.

With this knowledge, we can evaluate the classical proof of the binary theorem directly, by assigning the operator $\mathcal{C}$ to be the algorithmic content of the rule of double-negation elimination. Moreover, we can utilize results of Plotkin to define more efficient double-negation translations than the one which we originally chose, with the knowledge that these more efficient translations are every bit as powerful as the original ones, and, in addition, produce extensionally equivalent programs.

References

[1] S. F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.

[2] R. Burstall and B. Lampson. A kernel language for abstract data types and modules. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France*, volume 173 of *Lecture Notes in Computer Science*, Berlin, June 1984. Springer-Verlag.

[3] R. L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[4] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184. Springer-Verlag, 1985.

[5] M. Felleisen. *The Calculi of $\lambda_v - CS$ conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[6] M. Felleisen, D. Friedman, E.Kohlbecker, and B. Duba. Reasoning with continuations. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 131–141, 1986.

[7] M. J. Fischer. Lambda-calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, volume 7 of *Sigplan Notices*, pages 104–109, 1972.

[8] H. Friedman. Classically and intuitionistically provably recursive functions. In Scott, D. S. and Muller, G. H., editor, *Higher Set Theory*, volume 699 of *Lecture Notes in Mathematics*, pages 21–28. Springer-Verlag, 1978.

[9] G. Kreisel. Mathematical significance of consistency proofs. *Journal Of Symbolic Logic*, 23:155–182, 1958.

[10] D. Leivant. Syntactic translations and provably recursive functions. *Journal Of Symbolic Logic*, 50(3):682–688, 1985.

[11] V. Lifschitz. Constructive assertions in an extension of classical mathematics. *Journal Of Symbolic Logic*, 47:359–387, 1982.

[12] P. Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[13] C. Murthy. *Extracting Constructive Content from Classical Proofs: Compilation and the Foundations of Mathematics.* PhD thesis, Cornell University, Department of Computer Science, 1990.

[14] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[15] R. Pollack. The theory of lego, 1989. Unpublished draft.

[16] J. C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur, la Programmation, Lecture Notes in Computer Science 19*, pages 408–23. NY:Springer-Verlag, 1974.

[17] S. Shapiro. *Epistemic and Intuitionistic Arithmetic*, pages 11–46. North-Holland, 1985.

[18] A. S. Troelstra, editor. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics.* Springer-Verlag, 1973.

[19] J. van Heijenoort, editor. *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931.* Harvard University Press, Cambridge, Massachusetts, 1967.

An algorithm for testing conversion in Type Theory

Thierry Coquand

INRIA and University of Göteborg/Chalmers

Preliminary version

Abstract

We prove the correctness and completeness of an algorithm for testing conversion in type theory. The method applies to Edinburgh LF.

Introduction

The goal of this note is to present a "modular" proof, for various type systems with η-conversion, of the completeness and correctness of an algorithm for testing the conversion of two terms. The proof of completeness is an application of the notion of logical relations, and what we present can be seen also as an attempt to extend the notion of logical relation to dependent types. This algorithm is the basis of the interactive type checker/theorem prover ALF.

In order to simplify the presentation we will limit ourselves to type theory with only Π, and first with no universes. That is, the proof we give applies exactly for Martin-Löf's logical framework (the reader is refered to [6] for a presentation of this framework). After some motivations, we present the algorithm, and then, the proof of its completeness. Its correctness is a corollary. We sketch at the end how to extend the proof for Edinburgh LF, and for a formulation of intuitionistic set theory (in the sense of [6]) with one universe and η-conversion.

1 Informal motivation

One algorithm for testing the conversion of two terms in presence of η-conversion, is to apply β-reduction and η-reduction until one reaches normal form, and then to compare syntactically the normal form. However, as pointed out by V. Breazu-Tannen and J. Gallier, there may be problems with η-reduction for testing conversion when one adds new conversion rules.

Here is an example. Suppose that we start with simply typed λ-calculus, with β, η-conversion, one ground type o, and one constant $a : o$, and that we add one constant $f : o \to o$, with the conversion rule $f(x) = a$. The terms f, and $\lambda x.a$ are then convertible, but they are syntactically distinct and in β, η-normal form. This example however does not apply directly to Type Theory with inductive types, and it would be interesting to find a more informative example.

Another remark about η-reduction is that it is a "non-lazy" algorithm. To check whether $\lambda x.M$ is an η-redex, we have to reduce M until we can check whether it gets the form $\mathbf{app}(N, x)$, where x does not occur in N.

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

The solution developed by V. Breazu-Tannen and J. Gallier is to use the notion of *long* β,η-normal form instead. That is, we do η-expansion instead of η-reduction. Then, in the example, f is no longer in normal form, and reduces to $\lambda x.f(x)$ which reduces to $\lambda x.a$. This makes sense only in a typed framework.

Here, we show that it is possible instead to use an "untyped" algorithm, which seems, at least in a first approach, easier to implement. The idea is to compute the weak head-normal form of the two terms (in an untyped way), and, in order to take care of η-conversion, in the case where one weak-head normal form is an abstraction $\lambda x.M$ and the other is N a variable or an application, to compare recursively $\mathbf{app}(N,x)$ and M.

This algorithm can be applied also for Authomath like system (and General Type Systems extended with η-conversion). But it is not semi-complete if the type system does not have the normalisation property. It is directly used for type-checking and proof-checking in Type Theory. The fact that we used untyped λ-abstraction is irrelevant here, and the argument developed thus applies to the case of a typed abstraction. This algorithm seems to be quite natural, and F. Pfenning told me it is essentially the one used in his implementation (with C. Eliott) of Elf.

The correctness proof of the algorithm proves at the same time (weak) normalisation. It will be interesting to see if the ideas developed for proving Church-Rosser for type theory with η-conversion independently of a normalisation proof can be applied for giving an alternative proof of correctness and completness.

2 Weak head-normal form

The syntax is the following:

$$M \ := \ x \mid \mathbf{app}(M,M) \mid \lambda x.M \mid (\Pi x\colon M)M$$

We will say that a term is *canonical* if, and only if, it is an abstraction or a product.

The notion of weak head-normal form is given by its operational semantics.

$$x \Rightarrow x$$

$$\lambda x.M \Rightarrow \lambda x.M$$

$$(\Pi x\colon A)M \Rightarrow (\Pi x\colon A)M$$

$$\frac{M \Rightarrow \lambda x.M_1 \quad M_1[N] \Rightarrow P}{\mathbf{app}(M,N) \Rightarrow P}$$

$$\frac{M \Rightarrow M_1}{\mathbf{app}(M,N) \Rightarrow \mathbf{app}(M_1,N)} \quad M_1 \text{ not canonical}$$

Definition: M has a weak head-normal form N if, and only if, $M \Rightarrow N$. M_1 and M_2 are weakly equivalent, notation $M_1 \simeq M_2$ if, and only if, M_1 and M_2 have identical weak head-normal forms.

It is important to note the difference between the relation $M_1 \simeq M_2$ and Kleene equality, which would be defined as: if M_1 (resp. M_2) has a weak head-normal form, then so has M_2 (resp. M_1) and they are identical. With the present definition, $M_1 \simeq M_2$ implies that both M_1 and M_2 have a weak head-normal form.

Remark: A given term has at most one weak head-normal form, that is the algorithm described by the relation $\Rightarrow$ is deterministic. Furthermore, a weak head-normal form that is not canonical is either a variable, or of the form $\mathbf{app}(f, M)$ where f is a weak-head normal form that is not canonical.

Lemma 1: If $\mathbf{app}(M, N)$ has a weak head-normal form, then M has a weak head-normal form.

Lemma 2: If $M_1 \simeq M_2$, and $\mathbf{app}(M_1, N)$ have a weak head-normal form, then $\mathbf{app}(M_1, N) \simeq \mathbf{app}(M_2, N)$.

Remark: This is false in general if $\mathbf{app}(M_1, N)$ has no weak head-normal form. For instance, with $\Delta = \lambda x.\mathbf{app}(x, x)$, we have that $\Delta \simeq \Delta$, but not that $\mathbf{app}(\Delta, \Delta) \simeq \mathbf{app}(\Delta, \Delta)$.

3 An algorithm for β, η-conversion

We define recursively when two terms M_1 and M_2 are "equivalent", notation $M_1 \Leftrightarrow M_2$. It means intuitively that both M_1 and M_2 are normalisable, and that the normal form of M_1 and the normal form of M_2 are η-convertible.

$M \Leftrightarrow N$ if, and only if, M has a weak normal form M_0, N has a weak normal form N_0 and the pair (M_0, N_0) is of one of the following form:

- (x, x),
- $(\lambda x.M_1, \lambda x.N_1)$ and $M_1 \Leftrightarrow N_1$,
- $(\mathbf{app}(M_1, M_2), \mathbf{app}(N_1, N_2))$ with $M_1 \Leftrightarrow N_1$ and $M_2 \Leftrightarrow N_2$,
- $((\Pi x{:}\, M_1)M_2, (\Pi x{:}\, N_1)N_2)$ with $M_1 \Leftrightarrow N_1$ and $M_2 \Leftrightarrow N_2$,
- $(\lambda x.M_1, N_0)$ with $M_1 \Leftrightarrow \mathbf{app}(N_0, x)$, where N_0 is not canonical,
- $(M_0, \lambda x.N_1)$ with $\mathbf{app}(M_0, x) \Leftrightarrow N_1$, where M_0 is not canonical.

It is clear from this definition that $\Leftrightarrow$ is symmetric. Furthermore, if $M_1 \simeq M_2$ and $M_2 \Leftrightarrow N$, then $M_1 \Leftrightarrow N$.

Lemma 3: If $\mathbf{app}(M_1, x) \Leftrightarrow \mathbf{app}(M_2, x)$, then $M_1 \Leftrightarrow M_2$. Conversely, if $a \Leftrightarrow b$, $a \Rightarrow M$, $b \Rightarrow N$, M and N are not canonical, then $\mathbf{app}(a, x) \Leftrightarrow \mathbf{app}(b, x)$.

Proof: Both M_1 and M_2 have a weak head-normal form, by lemma 1. The proof is by cases on these weak head-normal form. It is direct if these weak head-normal forms are both abstraction, or are both terms that are not canonical. If M_1 reduces to N_1, which is an application, and M_2 reduces to $\lambda x.N_2$, then we have $\mathbf{app}(M_2, x) \simeq N_2$ and so we get that $\mathbf{app}(N_1, x) \Leftrightarrow N_2$, hence the result.

For the converse, notice that $M \Leftrightarrow N$. Since M, N are not canonical, $\mathbf{app}(a, x) \Rightarrow \mathbf{app}(M, x)$, and $\mathbf{app}(b, x) \Rightarrow \mathbf{app}(N, x)$. Since $x \Leftrightarrow x$, we get by definition $\mathbf{app}(M, x) \Leftrightarrow \mathbf{app}(N, x)$, and so $\mathbf{app}(a, x) \Leftrightarrow \mathbf{app}(b, x)$.

Lemma 4: $\Leftrightarrow$ is a partial equivalence relation, i.e. is symmetric and transitive.

Proof: By induction on the proof that $M_1 \Leftrightarrow M_2$, we show that if $M_2 \Leftrightarrow M_3$, then $M_1 \Leftrightarrow M_3$. This is clear if M_2 has a canonical weak-head normal form, and in the case where the weak-head normal form of M_2 is a variable. If M_2 has a weak-head normal form which is an application, then it is direct by induction if the weak-head normal form of M_1 and M_3 are both canonical (and abstractions) or both non-canonical. In the remaining symmetric two cases, the conclusion follows from the second part of lemma 3.

4 Computability relation

The algorithm above should be an algorithm for testing the conversion of two well-typed terms of the same type, that is, if $M_1, M_2 : A$, then we expect to have $M_1 \Leftrightarrow M_2$ if, and only if $M_1 = M_2 : A$. In particular, we have to prove that if M is well typed, then M has a weak head-normal form. It is natural for this to use the computability method, or its generalisation in the form of logical relation. What follows can be seen as a basic example of the use of logical relation with dependent types.

We recall (cf. [6]) that type theory can be seen as the inductive definition of two relations $A = B$ (meaning that A and B are equal types), and $a = b : A$ (meaning that a, b are equal elements of the type A) over a class of syntactic expression (the terms as described in the 2nd paragraph). The predicate A is a type can be seen as an abreviation of $A = A$, and the relation $a : A$ as an abreviation of $a = a : A$.

We define recursively when the computability relation $E_{A,B}$ is defined, and then what is its definition. Intuitively, $E_{A,B}$ is defined only when A and B are equal types, and then it represents the equality relation on the type A (which is then equivalent to the one on the type B).

Notation: For any type A, we introduce the relation $T_A(x, y)$ over syntactic objects x, y which holds if, and only if, $x = y : A$ and $x \Leftrightarrow y$.

Notice that if $A = B$, then T_A and T_B are equivalent relations.

Definition: $E_{A,B}$ is defined if, and only if, A, B are types such that A has a weak normal form A_0, B has a weak normal form B_0, and

- either both A_0, B_0 are not product, and $A = B$, and $A \Leftrightarrow B$. Then $E_{A,B}$ is defined as T_A (which is equivalent to T_B),
- or, A_0 is $(\Pi x\colon A_1)A_2$, B_0 is $(\Pi x\colon B_1)B_2$ and $A = (\Pi x\colon A_1)A_2$, $B = (\Pi x\colon B_1)B_2$, and E_{A_1,B_1} is defined, and $E_{A_1,B_1}(a_1, b_1)$ implies that $E_{A_2[a_1],B_2[b_1]}$ is defined. Then $E_{A,B}(a, b)$ holds if, and only if, $a = b : A$ and $E_{A_1,B_1}(a_1, b_1)$ implies $E_{A_2[a_1],B_2[b_1]}(\mathbf{app}(a, a_1), \mathbf{app}(b, b_1))$.

Remark: Notice that it is not clear a priori that if A is a type and $A \Rightarrow B$, then $A = B$. This will actually be a corollary of the proofs below.

What we are going to show is that, if $A = B$, then $E_{A,B}$ is defined and is then equivalent to T_A, and that if $x = y : A$, then $T_A(x, y)$, and hence $x \Leftrightarrow y$, which will show the completness of the algorithm $\Leftrightarrow$.

Let us first give a brief "justification" of the definition of $E_{A,B}$, following [1] (this was pointed out and explained to the author by S. Hayashi). The problem with the definition of $E_{A,B}$ is the fact that E occurs negatively in the inductive definition. We solve this by defining not a partial function $(A, B) \longmapsto E_{A,B}$, but a *functional relation* $R(A, B, E)$ which represents this partial function (i.e. we can read $R(A, B, E)$ as "E is the relation $E_{A,B}$"). The basic remark is then that this functional relation $R(A, B, E)$ between pairs of syntactic terms and relations E on syntactic objects can be defined with a positive inductive definition.

Notations: For A a type, E_1 a binary relation over syntactic objects and E_2 a 4th-ary relation over syntactic objects, let us write $\mathcal{F}(A, E_1, E_2)$ the binary relation on syntactic object that holds on (x, y) if, and only if, $x = y : A$ and $E_1(u, v)$ implies that $E_2(u, v, \mathbf{app}(x, u), \mathbf{app}(y, v))$. Furthermore, let us write $E \equiv F$ for expressing the fact that the two binary relation over syntactic objects E and F are extensionally equivalent, that is $E(x, y)$ if, and only if, $F(x, y)$ for all syntactic expressions x and y.

We can then present the definition of R as the least relation such that $R(A, B, E)$ holds if, and only if, either

- $A \Rightarrow A_0$, $B \Rightarrow B_0$ and A_0, B_0 are not products, and,
- $A = B$ and $A \Leftrightarrow B$, and,
- $E \equiv T_A$,

or, for some relations $E_1(u, v)$, $E_2(u, v, x, y)$,

- $A \Rightarrow (\Pi x : A_1)A_2$, $B \Rightarrow (\Pi x : B_1)B_2$, and,
- $A = (\Pi x : A_1)A_2$, $B = (\Pi x : B_1)B_2$, and,
- $R(A_1, B_1, E_1)$, and,
- $E_1(u, v)$ implies that $R(A_2[u], B_2[v], E_2(u, v))$, and,
- $E \equiv \mathcal{F}(A, E_1, E_2)$.

From the fact that the relation $\Rightarrow$ is deterministic, follows by induction that $R(A, B, E)$ is a functional relation, that is if $R(A, B, E)$ and $R(A, B, F)$, then $E \equiv F$.

It will be interesting to know what proof-theoretic strength is needed for being able to get this inductively defined relation $R(A, B, E)$.

Remark: It should be noticed that all we have done is to reduce the existence of the partial function $E_{A,B}$ to the existence of the positively inductively defined relation R. There is an impredicative interpretation of the relation R, since R is defined by a positive inductive definition. It is likely however that we can justify the existence of R in predicative theories. A possibility is to admit as an axiom the existence of the partial function $E_{A,B}$ (which is probably weaker than the use of general positive inductive definition).

Lemma 5: If $E_{A,B}$ is defined, then

- $A = B$, $A \Leftrightarrow B$ and, if $E_{A,B}(a, b)$, then $T_A(a, b)$,
- If $a = b : A$, and $a \Rightarrow M$, $b \Rightarrow N$, M and N are not canonical, and $M \Leftrightarrow N$, then $E_{A,B}(a, b)$.

Remark: In particular, if $E_{A,B}$ is defined, and x is a variable of type A, then $E_{A,B}(x,x)$. Indeed, this follows from the second part of lemma 5 and the fact that $x = x : A$, and $x \Leftrightarrow x$.

Proof: We prove this simultaneously by induction on the proof that $E_{A,B}$ is defined.

It is direct if both A and B have weak head-normal form that are not products, for then $E_{A,B}$ is defined as T_A.

Otherwise, $A \Rightarrow (\Pi x\colon A_1)A_2$, $B \Rightarrow (\Pi x\colon B_1)B_2$, $A = (\Pi x\colon A_1)A_2$, $B = (\Pi x\colon B_1)B_2$, and E_{A_1,B_1} is defined, and $E_{A_1,B_1}(a_1,b_1)$ implies that $E_{A_2[a_1],B_2[b_1]}$ is defined. By induction hypothesis, we have that $A_1 = B_1$, $A_1 \Leftrightarrow B_1$ and also that $E_{A_1,B_1}(x,x)$ if x is a fresh variable of type A_1. But then, we get that $E_{A_2[x],B_2[x]}$ is defined, hence, by induction, that $A_2[x] = B_2[x]$ and $A_2[x] \Leftrightarrow B_2[x]$. Thus, we have that $A = B$ and $A \Leftrightarrow B$. Furthermore, if $E_{A,B}(a,b)$, then $a = b : A$ by definition and since $E_{A_1,B_1}(x,x)$, we have $E_{A_2[x],B_2[x]}(\mathbf{app}(a,x),\mathbf{app}(b,x))$. Hence, by induction, $\mathbf{app}(a,x) \Leftrightarrow \mathbf{app}(b,x)$. By lemma 3, this implies $a \Leftrightarrow b$, hence $T_A(a,b)$. Finally, if $a = b : A$, if $a \Rightarrow M$, $b \Rightarrow N$, and M, N are not canonical and $M \Leftrightarrow N$, then $\mathbf{app}(a,a_1) \Rightarrow \mathbf{app}(M,a_1)$ and $\mathbf{app}(b,b_1) \Rightarrow \mathbf{app}(N,b_1)$, and $\mathbf{app}(a,a_1) = \mathbf{app}(b,b_1) : A_2[a_1]$, and $\mathbf{app}(M,a_1) \Leftrightarrow \mathbf{app}(N,b_1)$ if $E_{A_1,B_1}(a_1,b_1)$ since then $a_1 \Leftrightarrow b_1$ by induction. By induction, we have that $E_{A_2[a_1],B_2[b_1]}(\mathbf{app}(a,a_1),\mathbf{app}(b,b_1))$, and thus $E_{A,B}(a,b)$.

Corollary: If $E_{(\Pi x A_1)A_2,(\Pi x B_1)B_2}$ is defined, then E_{A_1,B_1} and $E_{A_2[x],B_2[x]}$ are both defined.

Proof: By definition of E, we get that E_{A_1,B_1} is defined, and that $E_{A_1,B_1}(a,b)$ implies that $E_{A_2[a],B_2[b]}$ is defined. By lemma 5, we have $E_{A_1,B_1}(x,x)$, if x is a fresh variable of type A_1, hence $E_{A_2[x],B_2[x]}$ is defined.

Remark: Lemma 5 can also be stated as follows. If $R(A,B,E)$, then $A = B$, $A \Leftrightarrow B$ and if $E(a,b)$, then $T_A(a,b)$. Furthermore, if $a = b : A$, $a \Rightarrow M$, $b \Rightarrow N$ and M,N are not canonical, and $M \Leftrightarrow N$, then $E(a,b)$. The proof of lemma 5 can then be seen as being done by induction on the proof of $R(A,B,E)$.

Lemma 6: If $E_{A,B}$ is defined, $E_{A,B}(a,b)$, $u = a : A$, $v = b : B$, and $a \simeq u$, $b \simeq v$, then $E_{A,B}(u,v)$.

Proof: By induction on the proof that $E_{A,B}$ is defined.

If A (resp. B) has a weak head-normal form A_0 (resp. B_0) which is not a product, then we have $A = B$, $A \Leftrightarrow B$, $a = b : A$ and $a \Leftrightarrow b$. Hence, we get $u = v : A$. Furthermore, $u \Leftrightarrow v$ by the remark following the definition of $\Leftrightarrow$.

Otherwise, $A \Rightarrow (\Pi x\colon A_1)A_2$, $B \Rightarrow (\Pi x\colon B_1)B_2$, $A = (\Pi x\colon A_1)A_2$, $B = (\Pi x\colon B_1)B_2$, and E_{A_1,B_1} is defined, and $E_{A_1,B_1}(a_1,b_1)$ implies that $E_{A_2[a_1],B_2[b_1]}$ is defined, and $E_{A_2[a_1],B_2[b_1]}(\mathbf{app}(a,a_1),\mathbf{app}(b,b_1))$. If we have $E_{A_1,B_1}(a_1,b_1)$, then, by lemma 5, $a_1 = b_1 : A_1$. This implies that $\mathbf{app}(u,a_1) = \mathbf{app}(a,a_1) : A_2[a_1]$, and $\mathbf{app}(v,b_1) = \mathbf{app}(b,b_1) : B_2[b_1]$. We have $E_{A_2[a_1],B_2[b_1]}(\mathbf{app}(a,a_1),\mathbf{app}(b,b_1))$, hence, by lemma 5, $\mathbf{app}(a,a_1) \Leftrightarrow \mathbf{app}(b,b_1)$, and so, both $\mathbf{app}(a,a_1)$ and $\mathbf{app}(b,b_1)$ have weak head-normal form. By lemma 2, this implies that $\mathbf{app}(u,a_1) \simeq \mathbf{app}(a,a_1)$, and $\mathbf{app}(v,b_1) \simeq \mathbf{app}(b,b_1)$. But we can then apply the induction hypothesis, and get that $E_{A_2[a_1],B_2[b_1]}(\mathbf{app}(u,a_1),\mathbf{app}(v,b_1))$, and so, by definition, $E_{A,B}(u,v)$.

Lemma 7: If $E_{A,B}$ is defined, then

- $E_{B,A}$ is defined, and $E_{A,B}$, $E_{B,A}$ are equivalent relations,
- furthermore, if $E_{B,C}$ is defined, then $E_{A,C}$ is defined, and $E_{A,B}$, $E_{B,C}$ and $E_{A,C}$ are equivalent relations.

Proof: By induction on the proof that $E_{A,B}$ is defined.

If A, B have weak head-normal form that are not products, then $A = B$, $A \Leftrightarrow B$, and hence $B \Leftrightarrow A$ by lemma 4, and $E_{B,A}$ is T_B which is equivalent to $E_{A,B}$ which is T_A. If $E_{B,C}$ is defined, then C has a weak head-normal form which is not a product, and $B = C$, $B \Leftrightarrow C$. Hence the result by lemma 4.

Otherwise, $A \Rightarrow (\Pi x\colon A_1)A_2$, $B \Rightarrow (\Pi x\colon B_1)B_2$, $A = (\Pi x\colon A_1)A_2$, $B = (\Pi x\colon B_1)B_2$, and E_{A_1,B_1} is defined, and $E_{A_1,B_1}(a_1,b_1)$ implies that $E_{A_2[a_1],B_2[b_1]}$ is defined, and $E_{A,B} \equiv \mathcal{F}(A, E_{A_1,B_1}, E_{A_2,B_2})$. That $E_{B,A}$ is defined, and is equivalent to $E_{A,B}$, is then clear by induction hypothesis. If furthermore, $E_{B,C}$ is defined, then $C \Rightarrow (\Pi x\colon C_1)C_2$, $C = (\Pi x\colon C_1)C_2$ and E_{B_1,C_1} is defined, and $E_{B_1,C_1}(b_1,c_1)$ implies that $E_{B_2[b_1],C_2[c_1]}$ is defined, and $E_{B,C} \equiv \mathcal{F}(A, E_{B_1,C_1}, E_{B_2,C_2})$. The result is then clear by induction hypothesis.

Remark: In particular, this implies that if $E_{A,B}$ is defined and $E_{A,B}(a,b)$, then $E_{A,A}$ is defined, and $E_{A,A}(a,b)$.

5 Completeness of the algorithm

Proposition 1: If $A = B$, then $E_{A,B}$ is defined, and if $a = b : A$, then $E_{A,A}$ is defined and $E_{A,A}(a,b)$.

As usual, we prove the more general assertions that if $x_1 : A_1, \ldots, x_n : A_n$ is a valid context, and $a_1, \ldots, a_n$, $b_1, \ldots, b_n$ are two instances of this context such that $E_{A_i[a_1,\ldots,a_{i-1}],A_i[b_1,\ldots,b_{i-1}]}$ is defined, and also such that we have $E_{A_i[a_1,\ldots,a_{i-1}],A_i[b_1,\ldots,b_{i-1}]}(a_i,b_i)$ for all i, then, if $A = B$ in this context, $E_{A[a_1,\ldots,a_n],B[b_1,\ldots,b_n]}$ is defined, and if $a = b : A$ in this context, then $E_{A[a_1,\ldots,a_n],A[b_1,\ldots,b_n]}$ is defined, and we have also that $E_{A[a_1,\ldots,a_n],A[b_1,\ldots,b_n]}(a[a_1,\ldots,a_n], b[b_1,\ldots,b_n])$.

We prove this by induction on the derivations of $A = B$, and $a = b : A$. The judgement A is a type is seen as an abreviation for $A = A$, and the judgement $a : A$ as an abreviation that $a = a : A$. As in [4], we can suppose that we have a constant in each type, and we can limit ourselves to closed terms. In order to alleviate the notational burden, we will not show explicitely the contexts.

- If the last rule is an equality rule, that is $a = b : B$ is derived from $A = B$ and $a = b : A$. Then, by induction, we know that $E_{A,B}$ is defined, that $E_{A,A}$ is defined, and that $E_{A,A}(a,b)$. But from the fact that $E_{A,B}$ is defined, by lemma 7, we can conclude that $E_{B,B}$ is defined, and is equivalent to $E_{A,A}$. Thus we have also that $E_{B,B}(a,b)$.

- If the last rule is a variable rule, then it is direct by hypothesis, and by the remark following lemma 5.

- If we have derived $(\Pi x\colon A_1)B_1 = (\Pi x\colon A_2)B_2$ from $B_1 = B_2\ \ [x : A_1]$ and $A_1 = A_2$, then, by induction hypothesis, we have that E_{A_1,A_2} is defined, and hence equivalent to E_{A_1,A_1} by lemma 7, and if $E_{A_1,A_1}(a_1,a_2)$, then $E_{B_1[a_1],B_2[a_2]}$ is defined. This implies that $E_{(\Pi x A)B_1,(\Pi x A)B_2}$ is defined.

- If we have derived $\lambda x.b_1 = \lambda x.b_2 : (\Pi x\colon A)B$ from $b_1 = b_2 : B\ [x : A]$ and A is a type, then, by induction hypothesis, we have that $E_{A,A}$ is defined, and that, if $E_{A,A}(a_1,a_2)$ is defined, then $E_{B[a_1],B[a_2]}$ is defined, and $E_{B[a_1],B[a_2]}(b_1[a_1], b_2[a_2])$. By definition, we get that $E_{(\Pi x A)B_1,(\Pi x A)B_2}$ is defined. Since $\mathbf{app}(\lambda x.b_1, a_1) = b_1[a_1] : B[a_1]$, $\mathbf{app}(\lambda x.b_2, a_2) = b_2[a_2] : B[a_2]$, $\mathbf{app}(\lambda x.b_1, a_1) \simeq b_1[a_1]$ and $\mathbf{app}(\lambda x.b_2, a_2) \simeq b_2[a_2]$, by lemma 6, we get that $E_{B_1[a_1],B_2[a_2]}(\mathbf{app}(\lambda x.b_1, a_1), \mathbf{app}(\lambda x.b_2, a_2))$. Hence, $E_{(\Pi x A)B_1,(\Pi x A)B_2}(\lambda x.b_1, \lambda x.b_2)$.

- If we have derived $\mathbf{app}(\lambda x.b, a) = b[a] : B[a]$ from $b : B\ [x : A]$ and $a : A$, then, by induction hypothesis, we have that $E_{A,A}$ is defined, that $E_{A,A}(a, a)$ and that $E_{A,A}(a_1, a_2)$ implies that $E_{B[a_1],B[a_2]}$ is defined and that $E_{B[a_1],B[a_2]}(b[a_1], b[a_2])$. We get thus that $E_{B[a],B[a]}$ is defined, and that $E_{B[a],B[a]}(b[a], b[a])$. Since $\mathbf{app}(\lambda x.b, a) = b[a] : B[a]$, and $b[a] = b[a] : B[a]$, and $\mathbf{app}(\lambda x.b, a) \simeq b[a]$, we can apply lemma 6, and we get that $E_{B[a],B[a]}(\mathbf{app}(\lambda x.b, a), b[a])$.

- If we have derived $c = \lambda x.\mathbf{app}(c, x) : (\Pi x\colon A)B$ from $c : (\Pi x\colon A)B$, then, by induction hypothesis, we have that $E_{(\Pi x A)B,(\Pi x A)B}$ is defined, and that $E_{(\Pi x A)B,(\Pi x A)B}(c, c)$. This is only the case if $E_{A,A}$ is defined, and that $E_{A,A}(a_1, a_2)$ implies that $E_{B[a_1],B[a_2]}$ is defined, and then $E_{B[a_1],B[a_2]}(\mathbf{app}(c, a_1), \mathbf{app}(c, a_2))$. We have $\mathbf{app}(c, a_2) = \mathbf{app}(\lambda x.\mathbf{app}(c, x), a_2) : B[a_2]$ and $\mathbf{app}(c, a_2) \simeq \mathbf{app}(\lambda x.\mathbf{app}(c, x), a_2)$ since $\mathbf{app}(c, a_2)$ has a weak head-normal form (which is a consequence of $E_{B[a_1],B[a_2]}(\mathbf{app}(c, a_1), \mathbf{app}(c, a_2))$ by lemma 5). By lemma 6, we get that $E_{B[a_1],B[a_2]}(\mathbf{app}(c, a_1), \mathbf{app}(\lambda x.\mathbf{app}(c, x), a_2))$, hence the result.

- If we have derived $\mathbf{app}(c, a_1) = \mathbf{app}(c, a_2) : B[a_1]$ from $c : (\Pi x\colon A)B$ and $a_1 = a_2 : A$, then, by induction, we have that $E_{(\Pi x A)B,(\Pi x A)B}$ is defined, that $E_{(\Pi x A)B,(\Pi x A)B}(c, c)$, that $E_{A,A}$ is defined and that $E_{A,A}(a_1, a_2)$. This implies, by definition, that $E_{B[a_1],B[a_2]}$ is defined, and that $E_{B[a_1],B[a_2]}(\mathbf{app}(c, a_1), \mathbf{app}(c, a_2))$. Hence the result by the remark following lemma 7.

Corollary: If $(\Pi x\colon A_1)A_2 = (\Pi x\colon B_1)B_2$, then $A_1 = B_1$ and $A_2[x] = B_2[x]$.

Proof: We get that $E_{(\Pi x A_1)A_2,(\Pi x B_1)B_2}$ is defined, hence by the corollary of lemma 5 that E_{A_1,B_1} is defined and that $E_{A_2[x],B_2[x]}$ is defined. By lemma 5, this implies $A_1 = B_1$ and $A_2[x] = B_2[x]$.

Remark 1: Another proof of this result, not based on logical relation, and which works for an arbitrary GTS, appears in [7], and in [3]. An important corollary of this result is that a type system with "conversion as judgements" is equivalent to a type system with "untyped conversion on well-typed terms". Another application is the equivalence between typed-reduction and untyped reduction on well-typed terms. There seems to be no direct proof of this equivalence.

Remark 2: The corollary can be seen directly for Martin-Löf's logical framework. However, it does not seem to be any direct argument neither for Edinburgh *LF*, nor for set theory with one universe. We shall see below that the same method used here extends directly to these more difficult cases.

Remark 3: The proof given here suggests strongly a formulation of Type Theory where the equality judgement has the form $A : a = b : B$, meaning that A, B are equal types, and that a, b are equal elements of these types. The assumptions have then the form $A_i : x_i = y_i : B_i$. Such a formulation appears in [5].

6 Correctness of the algorithm

Lemma 8: If $\lambda x.b : (\Pi x\colon A)B$, then $b : B\ [x : A]$.

Proof: We have B_1, A_1 such that $b : B_1\ [x : A_1]$, and that $(\Pi x\colon A)B = (\Pi x\colon A_1)B_1$. But then, $A = A_1$, and $B = B_1$ by the corollary of proposition 1, hence the result

Lemma 9: If A is a type and $A \Rightarrow B$, then $A = B$. Similarly, if $a : A$ and $a \Rightarrow b$, then $a = b : A$.

Proof: Follows by induction from lemma 8.

What follows is a weakning of the uniqueness of types, that does not hold in general since abstractions are untyped.

Lemma 10: If $a : A_1$, $a : A_2$ and $a \Rightarrow M$, where M is not canonical, then $A_1 = A_2$.

Proof: By lemma 9, we have that $M : A_1$ and $M : A_2$. We can then reason by induction on M. If M is a variable, we have $A_1 = A_2$, and both are convertible to the type of the variable M given by the context. If M is an application $\mathbf{app}(N, P)$, then we have $N : (\Pi x\colon B_1)C_1$, $P : B_1$, and $N : (\Pi x\colon B_2)C_2$, $P : B_2$, and $A_1 = C_1[P]$, $A_2 = C_2[P]$. But N is not canonical, hence, by induction, we have that $(\Pi x\colon B_1)C_1 = (\Pi x\colon B_2)C_2$. By the corollary of the proposition 1, this implies that $B_1 = B_2$ and $C_1 = C_2$, hence the result.

Proposition 2: If A is a type, B is a type, and $A \Leftrightarrow B$ then $A = B$. Similarly, if $a : A$, $b : A$ and $a \Leftrightarrow b$, then $a = b : A$. Furthermore, if $a \Leftrightarrow b$, $a : A$, $b : B$, and $a \Rightarrow M$, $b \Rightarrow N$, M and N not canonical, then $A = B$, and $a = b : A$.

This is proved by induction on the proof that $a \Leftrightarrow b$.

Remark: it appears from the analysis of the proofs given here that an important property of a term a is: $a \Rightarrow M$ where M is not canonical (for which the unicity of types holds), and an important relation between terms a and b is: $a \Rightarrow M$, $b \Rightarrow N$, $M \Leftrightarrow N$ and M, N are not canonical.

7 Extension to stronger theories

It seems possible to extend the proof in the case where there is a dependent sum of types, and surjective pairing. The only changes are in the notion of weak head-normal form and the definition of $\Rightarrow$ and $\Leftrightarrow$.

Let us sketch how to extend the proof given here to the Edinburgh LF. We will use the equational presentation of LF described in [2]. We need to extend the syntax with kinds Type, and $(\Pi x\colon A)K$ for A type, and $K[x]$ a kind, under the hypothesis $x : A$. We introduce, besides the partial function $E_{A,B}$ defined as above for A, B equal types, a partial function $F_{K,L}$, defined for K, L equal kinds. $F_{K,L}$ is defined if, and only if,

- either both K and L are the kind Type, and then $F_{K,L}(A, B)$ is "$E_{A,B}$ is defined",
- or, K is $(\Pi x\colon A)K_1$, L is $(\Pi x\colon B)L_1$ and and $E_{A,B}$ is defined, and $E_{A,B}(a, b)$ implies that $E_{K_1[a],L_1[b]}$ is defined. Then $F_{K,L}(M, N)$ holds if, and only if, $M = N : K$, that is, M, N are equal type family, and $E_{A,B}(a, b)$ implies $E_{K_1[a],L_1[b]}(\mathbf{app}(M, a), \mathbf{app}(N, b))$.

It is also possible to extend the proof to a type system with universes (as it is formulated in [4], but with conversion as judgements and with η-conversion). This corresponds to the extension of Edinburgh LF where one allows the formation of kinds of the form $(\Pi x\colon K_1)K_2[x]$, where K_1 and $K_2[x]$ $[x : K_1]$ are kinds. We define a new relation $F_{A,B}$ for the equality over "large types". The definition is similar to the one of $E_{A,B}$, with the only addition of the following clause: if $A \Rightarrow U_0$ and $B \Rightarrow U_0$, then $F_{A,B}$ is defined and $F_{A,B}(T_1, T_2)$ is "E_{T_1,T_2} is defined".

Remark: This argument emphasizes the apparent similarity (but also the crucial difference) between the formulation of intuitionistic set theory in Martin-Löf's framework [6] and intuitionistic set theory with one universe (which is similar to Edinburgh logical framework extended with a product over kinds). The use of types, and type variables, in Martin-Löf's framework gives a *conservative* extension of intuitionistic set theory. The introduction of the framework has for main purpose a more flexible formulation of intuitionistic set theory.

Conclusion

We tried to present a direct, semantically motivated, proof of the correctness and completness of an algorithm that tests conversion in type theory. Our proof can be seen as an expression of one possible semantics of type theory, and it applies also to Edinburgh LF, or to set theory with one universe and β, η-conversion. It may be interesting to try to simplify it for the case of Martin-Löf's framework.

References

[1] S. Allen. "A Non Type-Theoretic Semantics for Type-Theoretic Language." Ph. D. Thesis, Cornell U., September 87.

[2] R. Harper. "An Equational Formulation of LF." LFCS Report Series, ECS-LFCS-88-67, 1988.

[3] D. Howe. "Equality In Lazy Computation Systems." Proceeding of LICS 89.

[4] P. Martin-Löf. "An Intuitionistic Theory of Types." Unpublished manuscript, 1972.

[5] P. Martin-Löf. "Syntax and Semantics of Mathematical Language." Notes written by P. Hancock, Oxford, 1975.

[6] B. Nordström, K. Petersson, J. M. Smith. "Programming in Martin-Löf Type Theory." Oxford Science Publications, 1990.

[7] A. Salvesen. "The Church-Rosser Theorem for LF with beta,eta-reductions." These proceedings, 1989.

Nederpelt's Calculus extended with a notion of Context as a Logical Framework

Philippe de Groote

Université Catholique de Louvain, Unité d'informatique

Abstract

We define an extended version of Nederpelt's calculus which can be used as a logical framework. The extensions have been introduced in order to support the notions of mathematical definition of constants and to internalize the notion of theory. The resulting calculus remains concise and simple, a basic requirement for logical frameworks. The calculus manipulates two kind of objects : texts which correspond to λ-expressions, and contexts which are mainly sequences of variable declarations, constant definitions, or context abbreviations. Basic operations on texts and contexts are provided. It is argued that these operations allow one to structure large theories. An example is provided.

1 Introduction

This paper introduces the static kernel of a language called DEVA [13]. This language, which has been developed in the framework of the ToolUse Esprit project, is intended to express software development mathematically. The general paradigm which was followed considered *development methods as theories* and *developments as proofs*. Therefore, the kernel of the language should provide a general treatment of formal theories and proofs.

The problem of defining a generic formal system is comparable to the one of defining a general computing language. While, according to Church's thesis, any algorithm can be expressed as a recursive function, one uses higher level languages for the actual programming of computers. Similarly, one could argue that any formal system can be expressed as Post productions, but to use such a formalism as a logical framework is, in practice, inadequate.

The criteria according to which the expressive power of a logical framework should be evaluated are mostly practical and subjective. One speaks of *natural representations* of logics, or expressions close to *ordinary mathematical presentation.*

Such requirements were at the origin of the AUTOMATH project [6]; we chose one of the simplest version of AUTOMATH, as a starting point for our language : Nederpelt's Λ [11].

Nederpelt's calculus, introduced in next section, is a λ-typed λ-calculus. Its expressive power, as a logical framework, is similar to Edinburgh LF [9]. The systematic use of typed λ-abstraction provides a smooth treatment of syntax, rules, and proofs.

The representation of a theory, in Nederpelt's Λ, amounts to a sequence of declarations of variables. This sequence is the environment in which proofs can be developed, but is not an object of the calculus. The notion of context we have added to Λ is intended to internalize the notion of environment. The goal is to provide simple means for structuring large collections of theories within the calculus.

This problem of structuring theories is not new in the case of software development. It has already been tackled in various ways by works on program specifications. Our approach is somewhat different, since it is merely syntactic. The meaning of the operations we provide on theories is not based on semantics, but explained in term of reduction relations.

2 Nederpelt's calculus

2.1 Definition

In Nederpelt's Λ, there is no syntactic difference between types and terms, and λ-abstraction is used to represent both functional terms and dependent types. For this reason, one rather speaks globally of the expressions of the calculus. These expressions are built from an infinite alphabet of variables and a single constant τ, according to the following formation rules: τ is an expression; any variable is an expression; if e_1, and e_2 are expressions, so are $[\,x : e_1\,]\,e_2$ (typed abstraction) and $(\,e_1\ e_2\,)$ (application). The notions of free and bound variables are generalized by adding the following clause to the usual ones : Any free occurrence of a variable x in e_1 is free in $[\,x_1 : e_1\,]\,e_2$ (even if $x \equiv x_1$).

β-redices and β-reduction are as usual[1]. Nevertheless, because of dependent types, contractions and substitutions may now be performed within type labels.

Among expressions, one distinguishes well-typed expressions, i.e. expressions which obey to some applicability conditions. To this end, it is necessary to assign types to expressions. This raises a problem for free variables which are a priori untyped. For this reason, the type of an expression will be defined according to some environment assigning a type to each free variable of the expression. An environment is thus a sequence of variable declarations. Let E denotes environments, the expression $E, x\!:\!e$ will denote the environment obtained by adding the declaration $x\!:\!e$ at the end of E. A partial selecting operation $\mathrm{sel}_x(E\,)$ is defined on environments as follows :

$$\begin{aligned} \mathrm{sel}_x(E, x\!:\!e\,) &\equiv e \\ \mathrm{sel}_x(E, x_1\!:\!e\,) &\equiv \mathrm{sel}_x(E\,) \quad \text{if } x \not\equiv x_1. \end{aligned}$$

The type of an expression e according to some environment E – in short : $\mathrm{type}\,(E\,|\,e)$ – is defined a priori, independently of the fact that e is well-typed or not. There is no type assigned to τ nor to the expressions whose head is τ, which are so to speak the proper types of the calculus.

$$\begin{aligned} \mathrm{type}\,(E\,|\,x) &\equiv \mathrm{sel}_x(E\,) \\ \mathrm{type}\,(E\,|\,[\,x\!:\!e_1\,]\,e_2) &\equiv [\,x\!:\!e_1\,]\,\mathrm{type}\,(E,\,x\!:\!e_1\,|\,e_2) \\ \mathrm{type}\,(E\,|\,(\,e_1\ e_2\,)) &\equiv (\,\mathrm{type}\,(E\,|\,e_1)\ e_2\,) \end{aligned}$$

The typing operator defined above allows to express applicability conditions and to define the notion of well-typed expression :

$$E \vdash \tau$$

[1]The calculus which is presented here is Nederpelt's Calculus without extensionality. As is well known [14], the language theory of the calculus with extensionality is much more complicated, but the main results are preserved.

$E \vdash x \quad$ if $\mathrm{sel}_x(E)$ is defined.

$$\frac{E \vdash e_1 \quad E, x\!:\!e_1 \vdash e_2}{E \vdash [\,x\!:\!e_1\,]\,e_2}$$

$$\frac{E \vdash e_1 \quad E \vdash e_2 \quad \mathrm{type}\,(E \,|\, e_1) =_\beta [\,x\!:\!\mathrm{type}\,(E \,|\, e_2)\,]\,e_3}{E \vdash (\,e_1\;e_2\,)}$$

$$\frac{E \vdash e_1 \quad E \vdash e_2 \quad e_1 =_\beta [\,x\!:\!\mathrm{type}\,(E \,|\, e_2)\,]\,e_3}{E \vdash (\,e_1\;e_2\,)}$$

It could seem strange, in the definition above, that there are no correctness conditions on environments. This is justified by the fact that we are interested in closed expressions only. In other words, the correct expressions of the calculus are the ones which are well-typed according to the empty environment.
The main properties of Λ, proof of which may be found in [11, 14], are the following :

Church-Rosser : if $e_1 =_\beta e_2$, then e_1 and e_2 have a common β-reduct[2];

Closure : if e_1 is a correct expression and if $e_1 \rhd_\beta e_2$, then e_2 is correct;

Strong-normalization : if e is correct, then there is no infinite sequence of β-contractions starting with e.

From these properties, one may infer that the property of being well-typed is decidable.

Λ may be used for representing logics in a way similar to the Endinburgh Logical Framework [9]. The examples developed for LF [1] are quite easily adaptable to Λ. For instance, the following declarations of Λ to the LF-signature correspond Σ_{PA} given in [9] :

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\iota & : & \tau & o & : & \tau \\
\\
0 & : & \iota & \mathrm{succ} & : & [\,x\!:\!\iota\,]\,\iota \\
+ & : & [\,x\!:\!\iota\,]\,[\,x\!:\!\iota\,]\,\iota & \times & : & [\,x\!:\!\iota\,]\,[\,x\!:\!\iota\,]\,\iota \\
\\
= & : & [\,x\!:\!\iota\,]\,[\,x\!:\!\iota\,]\,o & < & : & [\,x\!:\!\iota\,]\,[\,x\!:\!\iota\,]\,o \\
\neg & : & [\,x\!:\!o\,]\,o & \wedge & : & [\,x\!:\!o\,]\,[\,x\!:\!o\,]\,o \\
\vee & : & [\,x\!:\!o\,]\,[\,x\!:\!o\,]\,o & \supset & : & [\,x\!:\!o\,]\,[\,x\!:\!o\,]\,o \\
\forall & : & [\,x\!:\![\,x\!:\!\iota\,]\,o\,]\,o & \exists & : & [\,x\!:\![\,x\!:\!\iota\,]\,o\,]\,o
\end{array}$$

Similarly, universal introduction and natural induction, for instance, are given as follows :

$$\begin{array}{lcl}
\mathrm{AI} & : & [\,p\!:\![\,x\!:\!\iota\,]\,o\,]\,[\,x\!:\![\,x\!:\!\iota\,]\,(\,p\;x\,)\,]\,(\,\forall\,p\,) \\
\mathrm{IND} & : & [\,p\!:\![\,x\!:\!\iota\,]\,o\,]\,[\,x\!:\!(\,p\;0\,)\,]\,[\,x\!:\![\,x\!:\!\iota\,]\,[\,y\!:\!(\,p\;x\,)\,]\,(\,p\,(\,\mathrm{succ}\;x\,)\,)\,]\,(\,p\;t\,)
\end{array}$$

There is no degree restriction in Λ. This means that any well-typed expression may be used as a type. For this reason, operators turning a term into a type, such as :

$$\mathrm{true} \;:\; [\,x\!:\!o\,]\,\tau$$

are not needed.

[2]In the non-extensional system, the Church-Rosser property holds for any pair of well-formed expressions; in the extensional one, it holds for correct expressions only.

2.2 Some limitations

Λ has not been designed for practical use. In [3], de Bruijn described how to consider a complete AUTOMATH book as a λ-term of a language called AUT-SL (AUTOMATH single line). This idea gave rise to Λ which was essentially used for language-theoretic studies [11, 14].

In [10], Nederpelt presents some translations of mathematical texts into Λ, and discusses translation difficulties.

Some difficulties are due to the type structure of Λ. For instance, Λ has uniqueness of types. Therefore, if types are used to represent sets, a natural number x will not be automatically a real number since N and R cannot be convertible expressions. Another example appears if one tries to use the abstraction $[\,x : a\,]\,b$ to represent the implication $a \supset b$. The following typing rule is not admissible in Λ :

$$\frac{a \;:\; o \qquad b \;:\; o}{[\,x\!:\!a\,]\,b \;:\; o}$$

where o would be the type of propositions. Such difficulties are raised when Λ is used by applying the *propositions as types* principle, they disappear when Λ is used as a logical framework, i.e. when applying the *judgements as types* principle.

Another difficulty is related to the handling of definitions. The only way of representing mathematical definitions in Λ is by means of β-redices $(\,[\,x\!:\!e_1\,]\,e_2\;e_3\,)$, where the operand e_3 represents the *definiens*, and the variable x the *definiendum*. Such redices, to be well-typed, must obey the following rule :

$$\frac{E \vdash e_1 \quad E \vdash e_3 \quad E,\, x\!:\!e_1 \vdash e_2 \quad \mathrm{type}\,(E \,|\, e_3) =_\beta e_1}{E \vdash (\,[\,x\!:\!e_1\,]\,e_2\;e_3\,)}$$

This rule is too strong. It requires that a mathematical text (e_2), in which some defined constant (x) occurs, must make sense (be well-typed) independently of the value assigned to the constant (e_3). For this reason, when translating an AUTOMATH book into Λ, it is necessary to eliminate all the definitions, by β-reduction. This partial normalization is, in practice, prohibitive. To solve this problem, de Bruijn designed the language $\Lambda\Delta$. The typing rules of $\Lambda\Delta$ are such that a redex, to be well-typed, must obey the following rule :

$$\frac{E \vdash e_1 \quad E \vdash e_3 \quad E \vdash [e_3/x]\,e_2 \quad \mathrm{type}\,(E \,|\, e_3) =_\beta e_1}{E \vdash (\,[\,x\!:\!e_1\,]\,e_2\;e_3\,)}$$

In the extended system which is introduced in this paper we combine both Λ and $\Lambda\Delta$: Nederpelt's conditions are kept for functional application, but β-redices are not used any more to represent definitions. Instead, we use an explicit mechanism :

$$[\,d := e_1\,]\,e_2$$

whose typing conditions are those of $\Lambda\Delta$.

A last practical drawback when using an expression of Λ to represent a mathematical text is that it gives rise to numerous repetitions inside the expression. The parts of the expression which are often repeated are not mere subexpressions, they are abstractors, pairs made of abstractors and operands, or sequences of these. Therefore, such repetitions cannot be avoided by using β-redices, they require some other abbreviation mechanism. Such a mechanism is proposed by Balsters who extends λ-calculus by introducing the notion of segment [2]. The tailor made notion of context we introduce in next section is a particular case of segment. While this notion is mainly introduced to internalize the notion of theory within the calculus, it may also serve to avoid the repetitions mentioned above.

3 Contexts

Let us consider some correct expression of Λ representing some proof performed, for instance, in Peano arithmetic. Such an expression starts with a sequence of abstractions corresponding to the signature and the rules of Peano system :

$$[\iota:\tau][o:\tau][0:\iota][\mathrm{succ}:[x:\iota]\iota][+:[x:\iota][x:\iota]\iota]\cdots$$

Hence, at first sight, the representation of a given theory withn the calculus corresponds to a sequence of abstractors. Since contexts are intended to represent theories, a context is, in first approximation, such a sequence, and we allow abstraction of such contexts :

$$[\iota:\tau\,;\,o:\tau\,;\,0:\iota\,;\,\mathrm{succ}:[x:\iota]\iota\,;\,+:[x:\iota][x:\iota]\iota\,;\,\cdots]\cdots$$

To turn contexts into objects of the calculus remains useless as long as we do not provide operations to manipulate them. To this end, we introduce context variables, and the possibility of abbreviating contexts :

$$s := [\iota:\tau\,;\,o:\tau\,;\,0:\iota\,;\,\mathrm{succ}:[x:\iota]\iota\,;\,+:[x:\iota][x:\iota]\iota\,;\,\cdots]$$

Such abbreviations must have a scope, hence they are themselves abstractors. At this point, we may refine our notion of context : a context is either a context variable bound to an abbreviation, or a sequence of abstractors. Abstractors, which are atomic contexts, are of three kinds : declarations, definitions, and context abbreviations.

Besides contexts, mere λ-expressions remain : we call them texts. A text is either the constant τ, a variable, a defined constant (i.e. a variable bound to a definition), the application of text to a text, or the abstraction of a context on a text.

Roughly speaking, to abstract a context on a text consists of stating the theory in which the text makes sense. Such an operation would be useful on contexts : it would amount to state the theory in which some subtheory or some enrichment would be developed. Therefore, we add to the calculus the possibility of abstracting a context on another context. Similarly, we allow the application of context on text. This operation serves to instantiate parametric contexts.

Figure 1 gives an example illustrating how contexts and operations on them are useful to structure the presentation of theories. The reader who can not guess the rules according to which this example is a well-typed text is invited to refer to next section.

This example must be understood as follows.

It begins by a large context which may be seen as a library of theories. This context contains, on the one hand, some fundamental declarations common to all the theories of the library, and on the other hand, subcontexts which correspond to different theories (theories of equality, of natural numbers, of lists, ...).

The theory of equality specifies an equivalence relation between the elements of a sort α. This sort α is a parameter of the theory. To specify the theories of natural numbers or of lists, equality between naturals or between lists is needed. Therefore, both theories import the theory of equality,

$$
\begin{array}{lll}
[& \text{prop} : \tau ; & \\
 & \text{sort} : \tau ; & \\
 & \vdots & \\
 & \text{equ-thy} := [& \alpha : \text{sort} ; \\
 & & = : [x{:}\alpha][x{:}\alpha]\,\text{prop} ; \\
 & & \text{REFL} : [x{:}\alpha]\, x = x ; \\
 & & \vdots \\
 & \quad] & ; \\
 & \text{nat-thy} := [& \text{nat} : \text{sort} \\
 & & \text{succ} : [x{:}\text{nat}]\,\text{nat} ; \\
 & & + : [x{:}\text{nat}][x{:}\text{nat}]\,\text{nat} ; \\
 & & \text{equ-thy}(\text{nat}) ; \\
 & & \text{CSUCC} : [n_1{:}\text{nat}][n_2{:}\text{nat}][x{:}n_1 = n_2](\text{succ}\ n_1) = (\text{succ}\ n_2) ; \\
 & & \vdots \\
 & \quad] & ; \\
 & \text{list-thy} := [& \alpha : \text{sort} ; \\
 & & =_\alpha : [x{:}\alpha][x{:}\alpha]\,\text{prop} ; \\
 & & \text{list} : [x{:}\text{sort}]\,\text{sort} ; \\
 & & \text{nil} : (\text{list}\ \alpha) ; \\
 & & \text{cons} : [a{:}\alpha][l{:}(\text{list}\ \alpha)](\text{list}\ \alpha) ; \\
 & & \text{equ-thy}(\text{list}\ \alpha) ; \\
 & & \text{CCONS} : \ [a_1{:}\alpha][a_2{:}\alpha][l_1{:}(\text{list}\ \alpha)][l_2{:}(\text{list}\ \alpha)] \\
 & & \qquad\qquad [x{:}a_1 =_\alpha a_2][x{:}l_1 = l_2]((\text{cons}\ a_1)\ l_1) = ((\text{cons}\ a_2)\ l_2) ; \\
 & & \vdots \\
 & \quad] & ; \\
 & \vdots & \\
][& \text{nat-thy} ; & \\
 & \text{list-thy}(\text{nat})(=)]\ \cdots &
\end{array}
$$

Figure 1: example

with the proper instantiation of the parameter α. This instantiation is achieved by application of context.

The theory of lists is also parametric. The parameter is the sort α to which the elements of the lists belong. To give the proper equality rules for lists, the equality between elements of sort α is needed. This relation ($=_\alpha$) is the second parameter of the theory.

After the library, one finds a second context. This context is the working context in which proofs (symbolized by the last three dots) are performed. This working context is built by importing some parts of the library. In the example, it specifies the theory of lists of natural numbers. Therefore, the theory of natural numbers is first imported, and then, in the scope of this importation, the theory of lists is imported with the appropriate instantiation.

4 Formal definition of the extended calculus

4.1 Formation rules

The expressions of the extended calculus are built out of an infinite alphabet of ξ-variables (variables bound to declarations), an infinite alphabet of δ-variables (variables bound to definitions), an infinite alphabet of σ-variables (variables bound to context abbreviations), and the single constant τ. Let x, d, and s range over ξ-, δ-, and σ-variables respectively. Let c and t range over contexts and texts. The formation rules are the following :

$$
\begin{array}{rcll}
c & ::= & x:t \ \mid\ d:=t \ \mid\ s:=[\,c\,] & (\textit{atomic contexts}) \\
 & \mid & s & (\textit{context variable}) \\
 & \mid & c\,;\,c & (\textit{abstraction on context}) \\
 & \mid & (\,c\,)(\,t\,) & (\textit{application of context}) \\
 & & & \\
t & ::= & \tau & (\textit{constant}) \\
 & \mid & x \ \mid\ d & (\textit{variables}) \\
 & \mid & [\,c\,]t & (\textit{abstraction on text}) \\
 & \mid & (\,t\ t\,) & (\textit{application of text}) \\
 & \mid & (\,t:t\,) & (\textit{judgement})
\end{array}
$$

Judgement is a simple construction we have not yet described. It allows to give explicitly the type of a text. This is useful, for instance, to give a typed definition, or intermediate results within a proof.

As it is, the above grammar is ambiguous : a context made of successive abstractions gives rise to different parse trees. This ambiguity is circumvented by considering these different parse trees to be equivalent. In other words, the syntactic equivalence between contexts, at the level of abstract syntax, is modulo associativity of context abstraction.

The use of context variables may give rise to problems when defining the notions of bound and free occurrences of a variable. On the one hand, there are the usual problems related to α-conversion; on the other hand, problems related to the possible use of free σ-variables. Consider, for instance, the following text :

$$[\,s:=[\,x:\tau\,;\,y:x\,]\,]\,[\,s\,]\,y$$

The last occurrence of y is bound to the declaration $y:x$ which occurs inside the context abbreviated by s. Consider the same example where the abbreviation has been removed :

$$[\,s\,]\,y$$

One needs further information about the σ-variable s in order to know whether the occurrence of y is bound or not.

These problems can be solved by using a name free notation for context variables [2] in the spirit of [5]. We consider that such a notation is used at the level of abstract syntax. Nevertheless, for the sake of readability, we use the concrete syntax above, which is sufficient for our purpose since we are interested in closed expressions only.

4.2 Environments and typing

The notion of environment has to be adapted to the extended calculus : an environment is a sequence of atomic contexts, i.e. a sequence of declarations, definitions, and abbreviations. Similarly, the operation of selection is extended :

$$\mathrm{sel}_X(E,\, x\!:\!t\,) \;\equiv\; t \quad \text{if } X \equiv x.$$

$$\mathrm{sel}_X(E,\, x\!:\!t\,) \;\equiv\; \mathrm{sel}_x(E\,) \quad \text{if } X \not\equiv x.$$

$$\mathrm{sel}_X(E,\, d := t\,) \;\equiv\; t \quad \text{if } X \equiv d.$$

$$\mathrm{sel}_X(E,\, d := t\,) \;\equiv\; \mathrm{sel}_x(E\,) \quad \text{if } X \not\equiv d.$$

$$\mathrm{sel}_X(E,\, s := [\,c\,]\,) \;\equiv\; c \quad \text{if } X \equiv s.$$

$$\mathrm{sel}_X(E,\, s := [\,c\,]\,) \;\equiv\; \mathrm{sel}_x(E\,) \quad \text{if } X \not\equiv s.$$

where X ranges over ξ-, δ-, and σ-variables.
To define the type of text, we need a way of turning a context into an environment. To this end, we define the partial normalization of a context, with respect to an environment, as follows :

$$\mathrm{pnc}\,(E\,|\,x\!:\!t) \;\equiv\; x\!:\!t$$

$$\mathrm{pnc}\,(E\,|\,d := t) \;\equiv\; d := t$$

$$\mathrm{pnc}\,(E\,|\,s := [\,c\,]) \;\equiv\; s := [\,c\,]$$

$$\mathrm{pnc}\,(E\,|\,s) \;\equiv\; \mathrm{pnc}\,(E\,|\,\mathrm{sel}_s(E\,))$$

$$\mathrm{pnc}\,(E\,|\,c_1\,;\,c_2) \;\equiv\; \mathrm{pnc}\,(E\,|\,c_1)\,;\,c_2$$

$$\mathrm{pnc}\,(E\,|\,(\,c\,)(\,t\,)) \;\equiv\; \text{if } \;\mathrm{pnc}\,(E\,|\,c) \equiv x\!:\!t_1\,;\,c_1 \;\; \begin{array}{ll} \text{then} & \mathrm{pnc}\,(E\,|\,[t/x]\,c_1) \\ \text{else} & \text{undefined} \end{array}$$

Thanks to this last definition, we may define the operation of pushing a context onto an environment :

$$E \oplus x\!:\!t \;\equiv\; E,\, x\!:\!t$$

$$E \oplus d := t \;\equiv\; E,\, d := t$$

$$E \oplus s := [\,c\,] \;\equiv\; E,\, s := [\,c\,]$$

$$E \oplus s \;\equiv\; E \oplus \mathrm{sel}_s(E\,)$$

$$E \oplus c_1\,;\,c_2 \;\equiv\; (E \oplus c_1) \oplus c_2$$

$$E \oplus (\,c\,)(\,t\,) \;\equiv\; E \oplus \mathrm{pnc}\,(E\,|\,(\,c\,)(\,t\,))$$

The typing relation is then a mere extension of the one introduced by Nederpelt :

$$\mathrm{type}\,(E\,|\,x) \;\equiv\; \mathrm{sel}_x(E\,)$$

$$\text{type}(E \mid d) \equiv \text{type}(E \mid \text{sel}_d(E))$$

$$\text{type}(E \mid [c]\, t) \equiv [c]\, \text{type}(E \oplus c \mid t)$$

$$\text{type}(E \mid (t_1\ t_2)) \equiv (\text{type}(E \mid t_1)\ t_2)$$

$$\text{type}(E \mid (t_1 : t_2)) \equiv \text{type}(E \mid t_1)$$

4.3 Conversion rules

The various constructions we have added to Nederpelt's calculus extend its practical expressive power. Nevertheless, they remain non essential in the sense that they do not extend the class of normal forms. To obtan this property we introduce reduction relations allowing to remove the different extensions. Because of the presence of definitions and abbreviations, these notions of reduction have to be defined with respect to environments.

In addition to β- (and possibly η-) redices, we introduce the following reducible expressions :

Reducible contexts :

$$E \vdash s \to \text{sel}_s(E)$$

$$E \vdash (x : t_1 ; c)(t_2) \to [t_2/x]\, c$$

Reducible texts :

$$E \vdash d \to \text{sel}_d(E)$$

$E \vdash [d := t_1]\, t_2 \to t_2$ if d does not occur free in t_2.

$E \vdash [s := [c]]\, t \to t$ if s does not occur free in t.

$$E \vdash [c_1 ; c_2]\, t \to [c_1][c_2]\, t$$

$$E \vdash (t_1 : t_2) \to t_1$$

Reduction is then defined as the least reflexive, transitive relation including the notions of reduction above, and obeying the following congruence rules :

$$\frac{E \vdash t_1 \to t_2}{E \vdash x : t_1 \to x : t_2} \qquad \frac{E \vdash t_1 \to t_2}{E \vdash d := t_1 \to d := t_2} \qquad \frac{E \vdash c_1 \to c_2}{E \vdash s := [c_1] \to s := [c_2]}$$

$$\frac{E \vdash c_1 \to c_2 \quad E \oplus c_1 \vdash c_3 \to c_4}{E \vdash c_1 ; c_3 \to c_2 ; c_4} \qquad \frac{E \vdash c_1 \to c_2 \quad E \vdash t_1 \to t_2}{E \vdash (c_1)(t_1) \to (c_2)(t_2)}$$

$$\frac{E \vdash c_1 \to c_2 \quad E \oplus c_1 \vdash t_1 \to t_2}{E \vdash [c_1]\, t_1 \to [c_2]\, t_2} \qquad \frac{E \vdash t_1 \to t_2 \quad E \vdash t_3 \to t_4}{E \vdash (t_1\ t_3) \to (t_2\ t_4)} \qquad \frac{E \vdash t_1 \to t_2 \quad E \vdash t_3 \to t_4}{E \vdash (t_1 : t_3) \to (t_2 : t_4)}$$

As usual, conversion $(=)$ is the transitive, symmetric closure of reduction.

4.4 Well-typedness

The well-typedness conditions for the extended calculus are a straightforward generalization of these introduced by Nederpelt. They are expressed by the following rules :

$$\frac{E \vdash t}{E \vdash x : t} \qquad \frac{E \vdash t}{E \vdash d := t} \qquad \frac{E \vdash c}{E \vdash s := [\,c\,]}$$

$E \vdash X$ if $\mathrm{sel}_X(E)$ is defined (where X ranges over σ-, ξ, and δ-variables).

$$\frac{E \vdash c_1 \quad E \oplus c_1 \vdash c_2}{E \vdash c_1\,;\,c_2}$$

$$\frac{E \vdash c \quad E \vdash t \quad c = x : \mathrm{type}\,(E \,|\, t)\,;\,c_1}{E \vdash (\,c\,)(\,t\,)}$$

$E \vdash \tau$

$$\frac{E \vdash c \quad E \oplus c \vdash t}{E \vdash [\,c\,]\,t}$$

$$\frac{E \vdash t_1 \quad E \vdash t_2 \quad \mathrm{type}\,(E \,|\, t_1) = [\,x : \mathrm{type}\,(E \,|\, t_2)\,]\,t_3}{E \vdash (\,t_1\; t_2\,)}$$

$$\frac{E \vdash t_1 \quad E \vdash t_2 \quad t_1 = [\,x : \mathrm{type}\,(E \,|\, t_2)\,]\,t_3}{E \vdash (\,t_1\; t_2\,)}$$

$$\frac{E \vdash t_1 \quad E \vdash t_2 \quad \mathrm{type}\,(E \,|\, t_1) = t_2}{E \vdash (\,t_1 : t_2\,)}$$

This extended system is such that the properties of Church-Rosser, closure and strong normalization are preserved, and the property of being well-typed is still decidable [7].

5 Evaluation

We argued, in the introduction, that the criteria according to which the expressive power of a logical framework must be evaluated are mostly practical and subjective. In our case, what are these criteria? What are the expected benefits?

To our mind, the main features of the calculus we have introduced in this paper are the following :

- it is an extension of Nederpelt's Λ;
- contrarily to de Bruijn's $\Lambda\Delta$, δ-redices (definitions of constants) and β-redices are kept disjoint;
- the approach to contexts is purely syntactic;

- the operations on contexts, i.e. abbreviation, abstraction, and application, remain basic.

These design choices have to be discussed according to practical considerations.

Extension of Nederpelt's Λ

As most of AUTOMATH languages, Λ is a λ-typed λ-calculus : functional terms and dependent types are identified. This characteristic goes so far in the case of Λ that there is no syntactic difference at all between terms and types : any well-typed term may act as a type, without any degree restriction.

This uniformity could sometimes be seen as a drawback. It could be argued that the identification between types and terms is purely syntactic, and that it obscures the structure of the calculus. One might agree with this criticism, when considering logical frameworks as formalisms intended to human communication.

Our standpoint is somewhat different : we consider our calculus as a "machine level language", i.e. a language into which higher level formalisms should be translated. From this point of view, the uniformity of Λ appears to be an advantage. Λ provides a unique set of data structures to handle formal languages as well as rules and proofs. For instance, the same type checking algorithm allows to decide whether a term is well-formed according to some signature, or whether a proof is correct according to some theory.

Distinction between β- and δ-redices

Since we consider our calculus as a kind of machine code, we must justify the distinction made between β- and δ-redices at this level.

Instantiation and definition are surely distinct concepts, but this does not imply that they must be represented by distinct means. They may both be represented as β-redices. Moreover, in the case of the representation of formal languages and theories, we just argued that it was economic to deal with a uniform construct. We must therefore provide further evidence.

A first argument concerns the structure of β-redices. A β-redex represents both a definition and its scope. The definition corresponds to the pair operand-abstractor. As such, it is hardly accessible as an object of the language. This fact would prevent us to turn definitions into atomic contexts. However, this argument is not definitive. Balsters' segments allow one to manipulate operand-abstractor pairs.

The second argument we give is more sensible. Theoretically, the type checking algorithm requires normalizations in order to decide the equality between operand types and operator domains. Practically, such normalizations are too costly, and it is necessary to provide heuristics trying to built common reducts far away from the normal forms.

Such heuristics take advantage of the distinction between β- and δ-redices. If β-redices are used for instantiation only, β-normalisation is not harmful. Experience shows that the oversized growth of normal forms is mainly due to δ-reduction while, in practice, the number of constants to be unfolded remains reasonable.

Similarly, higher order resolution involved a unification process which works on head normal forms. While theoretically too complex, this process is acceptable in practice if unification is not modulo δ-conversion [12].

Syntactic approach to contexts

The goal we followed by defining contexts was to internalize the notion of theory. To this end we defined contexts as constructs of the language and gave them a meaning in term of reduction relations. Other approaches would have been possible : semantics based approaches or type theoretical approaches.

Semantic and syntactic approaches are not exclusive, they are complementary. Nevertheless, with respect to the goal we followed, syntax comes first : we are interested in derivability, rather than in validity. For instance, if we use, in a given context, some induction principle, it will be because the principle is explicitly available as a rule, and not because our semantics would be based on initial algebras as it is the case for some specification languages.

A type theoretical approach would consist in enriching the type structure of the language, by providing existential types, for instance, in order to get types corresponding to theories. Such an enrichment would have destroyed Nederpelt's Λ uniformity. Terms belonging to existential types can hardly be considered as types.

Our goal was to provide a machine level framework, and, in this respect, we will say that we followed a data structure oriented approach. Nevertheless, further work should clarify the relation between the different approaches and provide possible models to contexts.

Basic operations on contexts

Finally, the operations we provide on contexts remain basic, especially when one compares them with operations available in higher level formalisms such as parametrization, instantiation, union, merging, importation, renaming...

Once more, this choice is justified by the fact we consider our calculus as a machine level language. For instance, the example of section 3 shows how parametrization, instantiation, and importation may be expressed by abstractions and applications. What we hope is to provide basic operations easy to implement and powerful enough to be composed into higher level operations.

Contexts have now to be evaluated with that respect : from the point of view of implementation, do they correspond to realistic data structure? from the point of view of utilization, may high level operations on theories be compiled in terms of basic operations on contexts? This evaluation process is under way. Prototypes supporting DEVA partially implement our notion of context [8]. The ToolUse project has provided non trivial examples of structuration of theories into contexts [15].

References

[1] A. Avron, F. Honsel, and I. Mason. Using typed lambda calculus to implement formal systems on a machine. In P. Dybjer, B. Nordström, K. Petersson, and J. Smith, editors, *Proceedings of the workshop on programming logic*, Report 37, University of Goteborg, 1987.

[2] H. Balsters. *Lambda Calculus extended with Segments.* PhD thesis, Technische hogeschool Eindhoven, 1986.

[3] N.G. de Bruijn. *AUT-SL, a single line version of AUTOMATH.* Memorandum 71-17, Department of Mathematics and Computing Science, Technische hogeschool Eindhoven, 1971.

[4] N.G. de Bruijn. Generalizing AUTOMATH by means of a lambda-typed lambda-calculus. In *Mathematical Logic and Theoretical Computer Science*, pages 71–92, Lecture Notes in pure and applied Mathematics, 106, Marcel Dekker, New York, 1987.

[5] N.G. de Bruijn. Lambda Calculus notations with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indigationes Mathematicae*, 34:381–392, 1972.

[6] N.G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606, Academic Press, 1980.

[7] Ph. de Groote. *Définition et Propriétés d'un métacalcul de représentation de théories.* Université Catholique de Louvain. In preparation.

[8] R. Gabriel. *ToolUse Project, Task S final Report.* GMD, Karlsruhe, 1989.

[9] R. Harper, F. Honsel, and G. Plotkin. A Framework for Defining Logics. In *Proceedings of the symposium on logic in computer science*, pages 194–204, 1987.

[10] R.P. Nederpelt. An Approach to theorem proving on the basis of a typed lambda-calculus. In *Proceedings of the 5th international conference on automated deduction*, pages 182–194, Lecture Notes in Computer Science, 87, Springer Verlag, 1980.

[11] R.P. Nederpelt. *Strong Normalization in a typed lambda calculus with lambda structured types.* PhD thesis, Technische hogeschool Eindhoven, 1973.

[12] L.C. Paulson. Natural Deduction as Higher Order Resolution. *Journal of logic programming*, 3:237–258, 1986.

[13] M. Sintzoff, M. Weber, Ph. de Groote, and J. Cazin. *Definition 1.1. of the generic Development Language DEVA.* Report RR 89-9, Université Catholique de Louvain, Unité d'Informatique, 1989.

[14] D.T. van Daalen. *The language theory of AUTOMATH.* PhD thesis, Technische hogeschool Eindhoven, 1980.

[15] M. Weber. Formalization of the Bird-Meertens algorithmic calculus in the DEVA meta-calculus. To appear in M. Broy and C. B. Jones, editors, *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.

Theo, an interactive proof development system

Joëlle Despeyroux

INRIA,
Sophia-Antipolis, 2004 Route des Lucioles,
F-06565 Valbonne Cedex, France
e-mail: joelle.despeyroux@mirsa.inria.fr

Abstract

Theo is an interactive, tactic-driven proof development system. The system builts proofs in a backward fashion, using a resolution rule. Proofs may be incomplete and may contain variable. Theo is a general theorem prover taking as input an object logic written by means of inference rules. Proof tactics are understood as belonging to the meta-logic, hence they are naturally implemented by means of inference rules. As a consequence, both the meta and the object levels of Theo are logics described in the programming language Typol.

Theo is implemented using the Centaur system, which results in two major advantages. First the theorem prover manipulates formulae as trees, hence it makes use of the tree-manipulating machinery that is the underpinning of the whole Centaur system. Second, the system Theo inherits naturally the graphic man-machine interface of Centaur. The system has been recently distributed to several sites, among those are the BRA sites from Edinburgh, Paris and Turin.

The system comes with a users's manual and two examples: Propositional Logic and the Calculus of Constructions. The demo will work through these two examples. The former example illustrates the case where the compact first order form chosen by Theo for the proof is conveniant. In the latter case, the Calculus of Constructions -given to the system by its type inference rules- is the logical framework in which object logics are described, as a signature. In this case, the proofs built by the theorem prover are only useful for user interface matters, including undoing a sub-proof. The advantage of the approach is here its generality. Proof refinement is a particular tactic whereas typing is a general tactic, which consists in 'evaluating' an expression $\rho \vdash m : t$. Constant definition is handled by the logical framework.

For the future, we think to introduce quantifiers and higher-order features in Typol, to describe some object logics -in particular first order logic- in a more elegant way.

[1]This research was partly supported by ESPRIT Basic Research Action 'Logical Frameworks".

A Proof Synthesis Algorithm for a Mathematical Vernacular

DRAFT

Gilles Dowek

INRIA*

Rocquencourt, France

dowek@margaux.inria.fr

Abstract

A Mathematical Vernacular is a language in which mathematics can be written, both usable for publishing and mathematically fully described. We present a step toward such a language that uses the Calculus of Constructions as a foundation.

The main idea is to get rid of proof-terms and write demonstrations of theorems as sequences of propositions, in such a way that a proof of each proposition is obvious when previous propositions are known. A proof-synthesis system is developed in order to check that a sequence of propositions is indeed a demonstration.

Introduction

In [10] we have pointed out the need of a Mathematical Vernacular i.e. a mathematical language both usable for publishing and mathematically fully described[1], and designed an elementary Vernacular. In this language we had to write proof terms, and we said that we wanted to get rid of them and write demonstrations as sequences of propositions in such a way that a proof of each proposition is "obvious" when previous proposition are known.

For instance if we have two theorems $th1 : P \to Q$ and $th2 : P$ and we want to prove Q, we do not want to write the proof-term $(th1\ th2)$ but rather write to a sentence such as "We know $P \to Q$ (th1) and P (th2) and so Q". The reader has to look up the propositions $P \to Q$ and P and find by himself the proof of Q.

For each lemma, the writer indicates only the statement and a list of those among the previous propositions which may be used to prove this lemma, and the reader has to find by himself the rule or the rules used in this proof. So the reader is doing proof synthesis while he reads. Thus proof synthesis is not only a way to avoid writing obvious and cumbersome details, it is also a way to avoid having an explicit grammar for proofs.

When such a demonstration is checked by a machine, a theorem prover must be used to verify that a proof of each lemma can indeed be found when previous lemmas are known. Thus the design of a Vernacular with proof synthesis, requires the solution of two problems, the first is to

*This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

[1] The expression *Mathematical Vernacular* has been created by N.G. de Bruijn, he defines it as *the very precise mixture of words and formulas used by mathematicians in their better moments, whereas the "official" mathematical language is taken to be some formal system that uses formulas only.*

design a theorem-prover and the second is to describe how the writer indicates to the reader (i.e. the theorem-prover) those among the previous lemmas which may be used, for instance with the phrases: "so", "using the two previous propositions", "using theorem (4)", "since we know $P \to Q$", etc. In this paper we focus only on the first problem.

Completeness, Termination and Power of the Theorem Prover

Because Predicate Calculus is undecidable, a theorem prover cannot be both complete and always terminating. Most of automatic theorem proving studies sacrifice termination to completeness. Two reasons can be found for this choice: first it is quite hard to settle a goal for an always terminating procedure because the simple minded procedure which always fails answers the problem and for any procedure there exists another one which is more powerful and still terminates. Then, when we have a complete method, it is always possible to get a terminating one, by giving a bound to the search time (or the deepness of the explored tree).

Since we want proof checking to be decidable, in the design of a Vernacular we need an always terminating algorithm, and thus this algorithm must be incomplete. Still, it must be powerful enough to make the Vernacular complete. This means that even if we cannot synthesize it, we must be able to break up each proof in small pieces that can be synthesized. Roughly speaking, it means that the transitive closure of the algorithm must be complete.

Then the more the theorem prover will be powerful, the more the Vernacular will be allusive. A second goal is to have a sufficiently allusive Vernacular. This goal is more difficult to appreciate because the "good" level of detail we want in a mathematical text is quite dependent on the authors. Nevertheless we have seen in [10] that with a sections mechanism (which permits to get rid of abstractions in proof-terms) we could always decompose a demonstration into little proof-terms of the form u or $(u\ v)$ whith u and v either symbolic names of already proved propositions or objects, i.e. terms whose type has type $Type$. Therefore we can get rid of proof-terms and write demonstrations as sequences of lemmas in giving as a proof of each a list of one or two terms which are symbolic names of known propositions, or objects. When such a proof is checked, the system considers the proof-term u or $(u\ v)$. Weather it is complete this Vernacular is too weak because:

- If we know the following propositions:

$$Trans : (x : T)(y : T)(z : T)(R\ x\ y) \to (R\ y\ z) \to (R\ x\ z)$$

$$u : (R\ t\ t')$$

$$v : (R\ t'\ t'')$$

  we need five lemmas to prove $(R\ t\ t')$ and would like this operation to be atomic.

- If we know the proposition: $u : (x : T)(R\ x)$ and we want to deduce $(R\ t)$ for $t : T$, we have to write u and t and we would like t to be found by unification between $(R\ x)$ and $(R\ t)$.

- Since some logical symbols (conjunction, disjunction, existential quantifier, contradiction and negation) are defined as abbreviations in the formalism itself, we need several lemmas to use a natural deduction rule as $\wedge$-elim and we would like this operation to be atomic.

- When we prove a proposition such as $P \to Q \to P$ we do not want to introduce explicitly the hypotheses P and Q but we want this to be done automatically.

Conventions

By term we mean a term of the Calculus of Constructions without universes [9] defined by the following typing rules, where K and K' are elements of $\{Prop, Type, Kind\}$.

$$\frac{}{[]\ well\ formed}$$

$$\frac{\Gamma \vdash T : K}{\Gamma @ [x : T]\ well\ formed}$$

$$\frac{\Gamma\ well\ formed}{\Gamma \vdash Type : Kind}$$

$$\frac{\Gamma\ well\ formed}{\Gamma \vdash Prop : Type}$$

$$\frac{\Gamma\ well\ formed \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : K \quad \Gamma @ [x : T] \vdash T' : K'}{\Gamma \vdash (x : T)T' : K'}$$

$$\frac{\Gamma \vdash T : K \quad \Gamma @ [x : T] \vdash T' : K' \quad \Gamma @ [x : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'}$$

$$\frac{\Gamma \vdash u : (x : T)T' \quad \Gamma \vdash v : T}{\Gamma \vdash (u\ v) : T'[x \leftarrow v]}$$

In this system the term $Kind$ has no type.

N.B.: In this system $Prop$ is $*$, $Type$ is $\Box$ and $Kind$ is Δ in Barendregt's Generalized Types System notation [1]. We could also consider a system with the rule $Prop : Kind$ instead of $Prop : Type$. In this system $Prop$ and $Type$ are two versions of $*$ and $Kind$ is $\Box$.

All the algorithms presented here work for both systems. The only difference is that when we know $w = Prop$ or $w = Type$ in the first system we can deduce that the type of w is $Type$ or $Kind$ and in the second system we know this type is $Kind$.

Except when we mention it, we consider equality between terms modulo $\beta\eta$-reduction, as representative of an equivalence class we take the β-normal, η-long form.

1 Informal presentation of the theorem proving method

We use an algorithm with a top-down *resolution* rule and an *introduction* rule like in Helmink [13].

1.1 Resolution

1.1.1 Instanciation and unification

When we have proved a proposition $u : (x : T)(y : T)(R\ x\ y)$, we can prove another one:

$$(u\ t\ t') : (R\ t\ t')$$

Some of the information given in the proof is redundant, t and t' can be found by unification between the goal $(R\ t\ t')$ and the head of the type of u[2]: $(R\ x\ y)$.

1.1.2 Generated goals

When there is in the context the following judgments:

$$Antisym : (x : T)(y : T)((R\ x\ y) \to (R\ y\ x) \to (Eq\ x\ y))$$

$$u : (R\ t\ t')$$

$$v : (R\ t'\ t)$$

we would like to deduce $(Eq\ t\ t')$ without intermediate lemmas.

So we unify the head of the proposition *Antisym*, $(Eq\ x\ y)$ with $(Eq\ t\ t')$. This provides a substitution $\{< x, t >, < y, t' >\}$. Then we instanciate the hypotheses, and get the propositions $(R\ t\ t')$ and $(R\ t'\ t)$. Then we look for proofs of these propositions. In this example proofs of the propositions are in the context: u and v, and we can build the proof-term $(Antisym\ t\ t'\ u\ v)$, but in general we have to try to unify these propositions with the heads of the types of the context, etc., till we get an empty set of goals. We generate that way a tree, if the set of goals of a node is empty the theorem is proved.

1.1.3 Implication and universal quantification

In the previous example we have made a distinction between variables universally quantified and hypotheses. This distinction is not relevant to the Calculus of Constructions where universal quantification and implication are both products. So we must deal in the same way with these two logical symbols. If we have in the context a judgement of type:

$$(x_1 : T_1)...(x_n : T_n)T$$

and a goal P unifies with T then a goal may be generated for each of the T_i, these goals can be of type *Prop*, *Type* or *Kind*.

1.1.4 Dependencies between goals

When we have in a context Γ a proposition that asserts that a relation is transitive:

$$Trans : (x : T)(y : T)(z : T)((R\ x\ y) \to (R\ y\ z) \to (R\ x\ z)),$$

[2] In these propositions all the constants are replaced by their values, a better method would be to replace constants by their values only when it is necessary.

and if the goal $(R\ t\ t'')$ is well formed in a context Γ, we generate three goals:

$$[T; (R\ t\ y); (R\ y\ t'')].$$

The first goal is the middle variable y used in transitivity. In the other two goals y is the solution of the first one. In order to express this dependencies, a variable must be associated to each goal:

$$[y : T; u : (R\ t\ y); v : (R\ y\ t'')]$$

1.1.5 Existential variables

In this example, the term $(R\ t\ y)$ is not well formed in the context Γ, but in the context $\Gamma@[y : T]$. In order to be able to express the goal $u : (R\ t\ y)$, we must declare $y : T$ in the context. Goals are therefore a special kind of variables, we will name them *existential variables*. By opposition, the ordinary variables are called *universal variables*.

Since goals are in the context, a proof synthesis problem is nothing more than a context that contains existential variables. In the previous example we have transformed the context:

$$\Gamma@[\exists x : (R\ t\ t'')]$$

into the context:

$$\Gamma@[\exists y : T; \exists u : (R\ t\ y); \exists v : (R\ y\ t'')]$$

The transformation of the first context into the second is associated to the substitution of $(Trans\ t\ y\ t''\ u\ v)$ to x. The solution of the three goals of the second context will be associated with substitutions that will bind y, u and v, and the composition of these substitutions will bind x to a term well formed in Γ.

When we have a context with an existential variable we can use, in order to solve it, only the types of the universal variables and of the constants of Γ.

Let us now explain how with the proposition:

$$Trans : (x : T)(y : T)(z : T)((R\ x\ y) \to (R\ y\ z) \to (R\ x\ z))$$

and the context:

$$\Gamma@[\exists x : (R\ t\ t'')]$$

we get the context:

$$\Gamma@[\exists y : T; \exists u : (R\ t\ y); \exists v : (R\ y\ t'')].$$

The first context ends with an existential variable of type $(R\ t\ t'')$. This type is unifiable with the head of the type of $Trans$, $(R\ x\ z)$. These two terms are well formed in the context:

$$\Gamma@[\exists x : T; \exists y : T; \exists z : T; \exists u : (R\ x\ y); \exists v : (R\ y\ z)]$$

Remark that it is this context which indicates those among the variables that can be instanciated in the unification process (existential variables) and those that cannot (universal variables)[3]. Let

[3]The usual terminology of unification is *variables* for existential variables and *constants* for universal variables. We prefer this one in order to avoid confusion with the constants of the Calculus of Constructions that are abbreviations.

$\sigma = \{< x, t >; < z, t'' >\}$ be a substitution solution of the unification of $(R\ x\ z)$ and $(R\ t\ t'')$. We apply σ to all the types of:

$$\Gamma@[\exists x : T; \exists y : T; \exists z : T; \exists u : (R\ x\ y); \exists v : (R\ y\ z)]$$

and we erase the declarations of x and z that are bound by the substitution σ and we get:

$$\Gamma@[\exists y : T; \exists u : (R\ t\ y); \exists v : (R\ y\ t'')].$$

In the general case, this operation is slightly more complex because higher order unification can introduce new existential variables:

Let x be the first variable of Γ which is linked by σ and $\Gamma = \Delta@[\exists x : P]@\Delta'$. The term $\sigma(x)$ is well formed in the context:

$$\Delta@[\exists y_1 : U_1; ...; \exists y_k : U_k]$$

where $y_1, ..., y_k$ are the new variables introduced by the unification. We build the context:

$$\Delta@[\exists y_1 : U_1; ...; \exists y_k : U_k]@\Delta''$$

where Δ'' is the list Δ' in which all the occurrences of x have been substituted by $\sigma(x)$. The context Γ is transformed that way till we get $\sigma(\Gamma)$.

1.1.6 Unification and pattern matching

In the method we describe here, we give a proposition scheme (i.e. a proposition that contains existential variables) and we look for the instances of that scheme that are provable. Generally a Resolution engine like Prolog gives the instances but not the proofs. Proofs can be computed by programming the deduction rules as Horn clauses and axioms as clauses reduced to a head, so that, if the proposition P is provable under this set of axioms, then $u : P$ (where u is an existential variable) has an instance which is provable under this set of Horn clauses and an instance of u is a proof of P (Helmink [13]). The drawback of this method, for our purpose, is that we introduce an existential variable u in the goal $u : P$ even if the term P is ground and we must always use a full unification procedure.

In a system where we give directly the proposition P to a theorem prover in which deduction rules are built in, we can restrict ourselves to a pattern matching procedure in all the (frequent) cases where P is ground. Of course, if we want a proof of this proposition, the computation of this proof must be built into the theorem prover.

1.1.7 Order in which goals are solved

When we have a context with several existential variables, for instance:

$$\Gamma@[y : T; u : (R\ t\ y); v : (R\ y\ t'')]$$

we can solve first the leftmost goal, $y : T$ or the rightmost one, $v : (R\ y\ t'')$.

If we solve the leftmost first, we always have ground goals: we solve first $y : T$, this gives a term t', then we solve $(R\ t\ t')$ and $(R\ t'\ t'')$. This method has two drawbacks: the first is that we have to solve goals of type $Type$, and we have to use all the judgements of the context, whereas we would like to use propositions only, the second is that this method is a little like choosing a y at random then looking if it works and trying another one if it does not.

This second drawback disappears if we solve the rightmost goal first, for instance if we have in the context a proposition $(R\ t'\ t'')$, we can use this proposition to find $y = t'$. On the other hand, we have to solve goals with existential variables.

In the use of the proof synthesis in a Vernacular, we can ask the writter to prove the two lemmas $(R\ t\ t')$ and $(R\ t'\ t'')$ before proving the theorem $(R\ t\ t'')$. When the proof synthesis system searches a proof of $(R\ y\ t'')$, there is the proposition $(R\ t'\ t'')$ in the context. The unification problem of $(R\ y\ t'')$ with $(R\ t'\ t'')$ is a pattern matching problem, but unusually, the ground term is the head of the known proposition and not the goal. So we need a pattern matching procedure that gives a complete set of unifiers if either of the two terms is ground.

Another method consists in using a complete unification procedure and generating goals with existential variables till all the existential variable are instanciated. Remark that between these two methods where goals are solved from the right to the left, there are many intermediate solutions in which we use an incomplete unification procedure that permits to solve some goals and where the user has to first prove lemmas corresponding to the others goals. The minimal specification of this procedure is that it must be complete at least if either of the two terms is ground.

1.1.8 Goals of type $Type$ and $Kind$

When we solve goals from the right to the left, we can hope that all the goals to be solved will be of type $Prop$ and that the ones of type $Type$ and $Kind$ will disappear before we try to solve them. Unfortunately, this is not always the case: these goals may occur for two reasons: first even if T is of type $Type$ or $Kind$, in $(x : T)P$, x may have no occurrence in P, then even if x has an occurrence in P, the unification with P' may not bind the variable x[4]. Thus we must, in order to synthesize these proofs, use all the types of Γ and not only the propositions.

Actually these goals of type $Type$ or $Kind$ do not seem to occur in well written proofs and we study them only for completeness.

1.2 Introduction of hypotheses

When we want to solve a context:

$$\Gamma@[\exists x : (y : T)C]$$

a method consists in solving the context:

$$\Gamma@[\forall y : T]@[\exists z : C]$$

We can this way prove propositions such as $P \to Q \to P$.

Nevertheless, in some cases, such propositions can only be proved directly. We have then to explore both cases, the one in which we introduce the hypothesis and the one in which we do not.

In this example, the universal variable y can be used to solve the existential variable z but not the other existential variables of Γ. This restriction can be expressed by saying that in order to solve an existential variable we can only use the types of universal variables and constants *which are declared before this existential variable.*

[4]For instance, if the goal is A and the head of the known proposition $(b\ x)$, the unification problem $(b\ x) = A$ has a solution $b = [y : T]A$ and x is not bound in that solution.

1.3 Premises

In the method presented above, when we look for a proposition or a type whose head unifies with the goal, we must look at the types of all universal variables and constants declared in the context before the goal. In general just a few of them are useful for the proof and we would like, in order to have an efficient proof synthesis method even if the context is big, to be able to help the theorem prover by giving it a subset π of Γ, *the premises*, in which it will look for these propositions.

We will first present the proof synthesis method without premises and then we will add this mechanism as a refinement.

2 Quantified contexts

2.1 Definitions

A *quantified variable declaration* is a triple (Q, x, T) (written $Qx : T$) where Q is a quantifier ($\forall$ or $\exists$), x a symbol and T is a term.

A *constant definition* is a triple (x, t, T) (written $x = t : T$) where x is a symbol, and t and T two terms.

A *quantified context* is a list of quantified variables declarations and constants definitions[5].

Let Γ be a quantified context, we write $Unquant(\Gamma)$ the (non quantified) context obtained by forgetting all the quantifiers.

Let Γ be a quantified context, Γ is *well formed* if $Unquant(\Gamma)$ is well formed.

Let Γ be a quantified context, t and T two terms, $\Gamma \vdash t : T$ if $Unquant(\Gamma) \vdash t : T$.

Let Γ be a well formed quantified context, and y a symbol, there is at most one item of Γ, $Qx : T$ or $x = t : T$ with $x = y$. If there is one, we say that y is be *declared* in Γ and if this item is of the form $x = t : T$ we say that y is a *constant* of Γ, if it is of the form $\forall x : T$, we say that y is an *universal variable* of Γ and if it is of the form $\exists x : T$,we say that y is an *existential variable* of Γ.

A quantified context Γ is said to be *universally quantified* if all the quantifiers of Γ are universal.

Remark: A (non quantified) context can be extended in only one way in a universally quantified context. We will therefore identify non quantified contexts and universally quantified contexts.

2.2 Well typed substitutions

Definition: Let Γ be a quantified context, E the set of the existential variables of Γ and σ a substitution such that $dom(\sigma) \subset E$. We define, by induction on the length of Γ, a relation of compatibility, σ *is well typed in* Γ, and if σ is well typed in Γ a context $\sigma(\Gamma)$.

- If $\Gamma = []$ then $\sigma = Id$, σ is well typed in Γ and $\sigma(\Gamma) = []$.

- If $\Gamma = \Delta @ [x = t : T]$ then if σ is well typed in Δ, then let $\Gamma' = \sigma(\Delta)$, we let $\sigma(\Gamma) = \Gamma' @ [x = \sigma(t) : \sigma(T)]$.

  In the other case σ is not well typed in Γ.

[5] These quantified contexts are very similar to those considered by Miller in [22].

- If $\Gamma = \Delta@[\forall x : T]$ then x is not in the domain of σ so $\sigma(x) = x$.

  If σ is well typed in Δ, then let $\Gamma' = \sigma(\Delta)$, if $\Gamma'@[\forall x : \sigma(T)]$ is well formed then σ is well typed in Γ and $\sigma(\Gamma) = \Gamma'@[\forall x : \sigma(T)]$.

  In the other case σ is not well typed in Γ.

- If $\Gamma = \Delta@[\exists x : T]$ then we let $\tau = \sigma - \{< x, \sigma(x) >\}$.

  If τ is well typed in Δ, then let $\Gamma' = \tau(\Delta)$ and $y_1 : U_1, ..., y_k : U_k$ be the variables of $\sigma(x)$ which are not declared in Γ' [6].

  If:

  $$\Gamma'@[\exists y_1 : U_1; ...; \exists y_k : U_k]$$

  is well formed and:

  $$\Gamma'@[\exists y_1 : U_1; ...; \exists y_k : U_k] \vdash \sigma(x) : \sigma(T)$$

  then σ is well typed in Γ and:

  $$\sigma(\Gamma) = \Gamma'@[\exists y_1 : U_1; ...; \exists y_k : U_k]$$

  In the other cases σ is not well typed in Γ.

Remark: Let Γ be a quantified context and τ and σ two substitutions, then $(\tau \circ \sigma)(\Gamma) = \tau(\sigma(\Gamma))$.
Proof: By induction on the length of Γ.

Remark: If $\Gamma@[x : T]@\Delta \vdash u : T'$ and $\Gamma \vdash t : T$ then $\Gamma@\Delta[x \leftarrow t] \vdash u[x \leftarrow t] : T'[x \leftarrow t]$.
Proof: By induction on the length of Δ.

Remark: Let Γ be a quantified context, σ a substitution well typed in Γ and t and T two terms. If $\Gamma \vdash t : T$ then $\sigma(\Gamma) \vdash \sigma(t) : \sigma(T)$.
Proof: Let $\Gamma = [d_1; ...; d_n]$ where d_i is the declares the variable or the constant x_i.
Let $\sigma_i = \sigma_{|\{x_1,...,x_i\}}$. Obviously, σ_0 is the identity and $\sigma_n = \sigma$.
We prove by induction that:

$$\sigma_i(\Gamma) \vdash \sigma_i(u) : \sigma_i(T)$$

It is obvious for $i = 0$. Let us assume this property for i.

- If d_{i+1} is a constant definition or a universal variable declaration then $\sigma_{i+1} = \sigma_i$ so :

  $$\sigma_{i+1}(\Gamma) \vdash \sigma_{i+1}(u) : \sigma_{i+1}(T)$$

- If d_{i+1} is an existential variable declaration then:

  $$\sigma_i([d_1; ...; d_i])@[\exists x_{i+1} : \sigma_i(T_{i+1})]@[\sigma_i(d_{i+2}); ...; \sigma_i(d_n)] \vdash \sigma_i(u) : \sigma_i(T)$$

  Let $y_1 : U_1; ...; y_k : U_k$ be the variables of $\sigma(x_{i+1})$ which are not declared in $\sigma_i([d_1; ...; d_i])$.
  Since it is always possible to add useless variables in a context, if their types are well formed:

  $$\sigma_i([d_1; ...; d_i])@[\exists y_1 : U_1; ...; \exists y_k : U_k]@[\exists x_{i+1} : \sigma_i(T_{i+1})]@[\sigma_i(d_{i+2}); ...; \sigma_i(d_n)] \vdash \sigma_i(u) : \sigma_i(T)$$

[6] Considering these variables $y_1, ..., y_k$, two important particular cases may be remarked:
If $\sigma(x) = x$ then $k = 1, y_1 = x, U_1 = \sigma(T)$.
If $\sigma(x)$ is a well formed term in Γ' then $k = 0$.

Since:

$$\sigma_i([d_1; ...; d_i])@[\exists y_1 : U_1; ...; \exists y_k : U_k] \vdash \sigma_i(x_{i+1}) : \sigma_i(T_{i+1})$$

and using the previous remark:
$\sigma_i([d_1; ...; d_i])@[\exists y_1 : U_1; ...; \exists y_k : U_k]@[\sigma_i(d_{i+2})[x_{i+1} \leftarrow \sigma(x_{i+1})]; ...; \sigma_i(d_n)[x_{i+1} \leftarrow \sigma(x_{i+1})]]$
$\vdash \sigma_i(u)[x_{i+1} \leftarrow \sigma(x_{i+1})] : \sigma_i(T)[x_{i+1} \leftarrow \sigma(x_{i+1})]$
Thus:

$$\sigma_i([d_1; ...; d_i])@[\exists y_1 : U_1; ...; \exists y_k : U_k]@[\sigma_{i+1}(d_{i+2}); ...; \sigma_{i+1}(d_n)] \vdash \sigma_{i+1}(u) : \sigma_{i+1}(T)$$

$$\sigma_{i+1}([d_1; ...; d_i; d_{i+1}])@[\sigma_{i+1}(d_{i+2}); ...; \sigma_{i+1}(d_n)] \vdash \sigma_{i+1}(u) : \sigma_{i+1}(T)$$

Therefore:

$$\sigma_{i+1}(\Gamma) \vdash \sigma_{i+1}(u) : \sigma_{i+1}(T)$$

Thus:

$$\sigma_n(\Gamma) \vdash \sigma_n(u) : \sigma_n(T)$$

i.e.:

$$\sigma(\Gamma) \vdash \sigma(u) : \sigma(T)$$

Definition: A substitution σ is said to be a *solution* of Γ is it is well-typed in Γ and $\sigma(\Gamma)$ is a universally quantified context.

Remark: Let Γ be a quantified context and σ and η two substitutions. $\eta \circ \sigma$ is a solution of Γ if and only if η is a solution of $\sigma(\Gamma)$.
Proof: $\eta \circ \sigma$ is a solution of Γ if and only if $\eta \circ \sigma(\Gamma)$ is a universally quantified context if and only if $\eta(\sigma(\Gamma))$ is a universally quantified context if and only if η is a solution of $\sigma(\Gamma)$.

Remark: σ is a solution of Γ if and only if for all existential variable x of Γ, $\sigma(\Gamma_{|x}) \vdash \sigma(x) : \sigma(T)$.
Proof: σ is a solution of Γ if and only if $\sigma(\Gamma)$ is a universally quantified context i.e. for all existential variable x, there are no new existential variable in $\sigma(x)$, i.e. for all existential variable x, $\sigma(\Gamma_{|x}) \vdash \sigma(x) : \sigma(T)$.

2.3 General form of a normal term in the Calculus of Constructions

Remark: Let t be a normal term. t can be written in an unique way:

$$t = [x_1 : T_1]...[x_n : T_n](y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$$

where w is a variable or one of the symbols $Prop$, $Type$ and $Kind$ (in which case $q = 0$).
Proof: The term t may be written in a unique way:

$$t = [x_1 : T_1]...[x_n : T_n]u$$

where u is not an abstraction. The term u is therefore a product, an application, a variable or one of the symbols $Prop$, $Type$ and $Kind$. It may be written:

$$u = (y_1 : U_1)...(y_p : U_p)v$$

where v is not a product. The term v is not an abstraction (if $p \neq 0$ because its type is $Prop$, $Type$ or $Kind$ and if $p = 0$ because u is not an abstraction). It is therefore an application, a variable or one of the symbols $Prop$, $Type$ and $Kind$. It may be written in a unique way:

$$v = (w\ s_1\ ...\ s_q)$$

where w is not an application. The term w is not a product (if $q \neq 0$ because a product is of type $Prop$, $Type$ or $Kind$ and therefore cannot be applied and if $q = 0$ because v is not a product). It is not an abstraction (if $q \neq 0$ because t is in normal form and if $q = 0$ because v is not an abstraction). It is therefore a variable or one of the symbols $Prop$, $Type$ and $Kind$.

Definition: Let t a normal term. It may be written in a unique way:

$$t = [x_1 : T_1]...[x_n : T_n](y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$$

The integer p is called *the number of initial products of* t.

Remark: Let t be a term which type is not a product, it may be written in a unique way:

$$t = (y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$$

Proof: The term t can be written:

$$t = [x_1 : T_1]...[x_n : T_n](y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q).$$

Since the type of t is not a product, t is not an abstraction and thus $n = 0$.

Definition: A term t is said to be atomic if it is equal to $(w\ s_1\ ...\ s_q)$ where w is a variable one one of the symbols $Prop$, $Type$ and $Kind$.

Remark: Let t be a term which type is not a product, t is a product or is atomic.
Proof: $t = (y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$. If $p \neq 0$ then t is a product, if $p = 0$ then t is atomic.

Remark: Let $t = [x_1 : T_1]...[x_n : T_n](y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$ and T such that $t : T$ in a context Γ then $w \neq Kind$.
Proof: by induction on the length of a derivation of $\Gamma \vdash t : T$.

2.4 Order of a type

We want to generalize the notion of order of a type of the simple theory of types to the Calculus of Constructions. We want to define this notion for a term T of type $Prop$, $Type$ or $Kind$ in a context Γ. This context may be quantified or not, but since non quantified contexts are identified with universally quantified contexts, it is sufficient to give the definition for quantified contexts.

The main problem in this definition is that if T is type variable (i.e. a variable of type $Prop$, $Type$ or $Kind$) we cannot let $o(T)$ be 1 because we want first order variables to be of null arity and T can be instanciated by any type, in particular by a functional type. We have therefore to distinguish between two cases: if the variable T is a universal variable, it cannot be instanciated, we let $o(T) = 1$, and if it is an existential variable or a variable bound higher in the type (see example $(U : Prop)(U \rightarrow U)$ below), it can be instanciated and we let $o(T) = \infty$.

We define the order of a type as a element of $N \cup \{\infty\}$, we extend arithmetics operations:

$$\forall n \in N \cup \{\infty\}, n + \infty = \infty + n = \infty$$

$$\forall n \in N \cup \{\infty\}, n \leq \infty$$

Definition: Let Γ be a quantified context and T be a normal term of type $Prop$, $Type$ or $Kind$. T is a product $T = (y : U)V$ or T is atomic $T = (w\ s_1\ ...\ s_q)$. The *order* of T is:

- if $T = (y : U)V$ then $o(T) = max\{1 + u, v\}$ where u is the order of U in the context Γ and v is the order of V in the context $\Gamma@[\exists y : U]$.
- if $T = (w\ s_1\ ...\ s_q)$ then if w is an universal variable of Γ then $o(T) = 1$, if w is an existential variable of Γ then $o(T) = \infty$ and if $w = Prop$, $w = Type$ or $w = Kind$ then $o(T) = 2$.

Examples: In the context $\Gamma = [\forall T : Type]$, we get:
$o(T) = 1$,
$o(T \to T) = o(T \to T \to T) = 2$,
$o((T \to T) \to T) = 3$,
$o((U : Prop)(U \to U)) = \infty$, because the order of U in $\Gamma@[\exists U : Prop]$ is infinite.
In the context $\Gamma' = [\exists T : Type]$ the order of all these types is infinite.

Remark: Preservation of the order with well-typed substitutions
Let Γ be a quantified context, T a type well formed in Γ and n the order of T in Γ. If σ is a well-typed substitution then $\sigma(T)$ has an order less or equal to n in $\sigma(\Gamma)$.
Proof: By induction on the structure of T.

- If $T = (w\ t_1\ ...\ t_n)$ then:
  - if w is universal in Γ then it is not instanciated by σ and it is still universal in Γ', $o(\sigma(T)) = o(T) = 1$,
  - if $w \in \{Prop, Type, Kind\}$ then $o(\sigma(T)) = o(T) = 2$,
  - if w is existential in Γ then $o(T) = \infty$ so $o(\sigma(T)) \leq o(T)$.
- If $T = (y : U)V$ then let us write $u' = o(\sigma(U))$ in $\sigma(\Gamma)$ and $v' = o(\sigma(V))$ in $\sigma(\Gamma)@[y : \sigma(U)] = \sigma(\Gamma@[y : \sigma(U)])$, by induction hypothesis $u' \leq o(U)$ and $v' \leq o(V)$ so $o(\sigma(T)) \leq o(T)$.

2.5 Ground terms and terms in which all free existential variables are at most of order 2

In simple type theory, universal variables cannot have existential variables in their types, so a ground term is only a term in which no existential variable has an occurrence. Here the universal variables must also have ground types.

Definition: Let Γ be a quantified context and $x : T$ a variable declared in Γ, we write $\Gamma_{|x}$ the unique context Δ such that $\Gamma = \Delta@[Qx : T]@\Delta'$.

Definition: Let Γ a quantified context and t a term well formed in Γ. We say that t is a *ground* term in Γ, if for all x free in t:

- x is universal,
- the type of x is ground in $\Gamma_{|x}$.

Definition: Let Γ be a quantified context, and t a term well formed in Γ, we say that *all the free existential variables of t are of order at most 2* if and only if for all x free in t:

- if x is existential then the type of x is of order at most 2 in $\Gamma_{|x}$,
- the type of x has all its free existential variables at most order 2.

3 A restriction to the Calculus of Constructions

The proof synthesis method we will present below uses a unification procedure or at least a pattern matching procedure. Since the Calculus of Constructions is a higher order formalism, the decidability of pattern matching in this language is still an open problem, but pattern matching is decidable if the non ground term has all its free variables of order at most 2.

So we will define a restriction of the Calculus of Constructions in which proof synthesis uses only this second order pattern matching. We want also this restriction to be powerful enough to express logical connectives, induction on inductive types and Leibnitz equality.

Definition: Let Γ be a quantified context and T a term of type $Prop$, $Type$ or $Kind$:

$$T = (y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$$

T is a *little type* in Γ if for all i:

- either U_i is of order at most 2 in $\Gamma@[\exists y_1 : U_1; ...; \exists y_{i-1} : U_{i-1}]$ or y_i has no occurrence in $(y_{i+1} : U_{i+1})...(y_p : U_p)(w\ s_1\ ...\ s_q)$
- U_i is a little type in $\Gamma@[\exists y_1 : U_1; ...; \exists y_{i-1} : U_{i-1}]$.

Example: the type $(P : bool \rightarrow Prop)(u : (P\ true))(v : (P\ false))(b : bool)(P\ b)$ is a little type, indeed $bool \rightarrow Prop$ is order 2, $(P\ true)$ and $(P\ false)$ have an infinite order but u and v have no occurrence, $bool$ is first order, and all these types are little types.

Definition: Let Γ be a quantified context, a term t is a *little term*, if its type is a little type.

Definition: A quantified context Γ is a *little quantified context* if all the types of the variables and constants declared in Γ are little types.

Remark: Let Γ be a little quantified context and $T = (x_1 : U_1)...(x_p : U_p)(w\ s_1\ ...\ s_q)$ a little type in Γ whose all free existential variables are of order at most 2, then:

- $\Gamma' = \Gamma@[\exists x_1 : U_1]$ is a little quantified context,
- $T' = (x_2 : U_2)...(x_p : U_p)(w\ s_1\ ...\ s_q)$ is a little type in Γ',
- all the free existential variables of T' are of order at most 2.

Proof:

- Γ' is a little context because Γ is a little context and U_1 is a little type in Γ.
- T is a little type, so for all $i \in [1,p]$:
  - either U_i is of order at most 2 in $\Gamma@[\exists y_1 : U_1; ...; \exists y_{i-1} : U_{i-1}]$ or y_i has no occurrence in $(y_{i+1} : U_{i+1})...(y_p : U_p)(w\ s_1\ ...\ s_q)$
  - U_i is a little type in $\Gamma@[\exists y_1 : U_1; ...; \exists y_{i-1} : U_{i-1}]$.

  So, for all $i \in [2,p]$:

  - either U_i is of order at most 2 in $\Gamma'@[\exists y_2 : U_2; ...; \exists y_{i-1} : U_{i-1}]$ or y_i has no occurrence in $(y_{i+1} : U_{i+1})...(y_p : U_p)(w\ s_1\ ...\ s_q)$
  - U_i is a little type in $\Gamma'@[\exists y_2 : U_2; ...; \exists y_{i-1} : U_{i-1}]$.

  So, T' is a little type in Γ'.
- Let y be a free variable of T':
  - If $y \neq x$ then y was already a free variable of T, its type V is therefore of order at most 2 in $\Gamma_{|y}$ and all its free variables are of order at most 2 in $\Gamma_{|y}$.
    Since $\Gamma'_{|y} = \Gamma_{|y}$, the same is true for $\Gamma'_{|y}$.
  - If $y = x$ then it means that x has an occurrence in T', so U_1 is at most order 2 in Γ, moreover since U_1 is a subterm of T, it has all its free variables are of order at most 2.

  T' has all its free existential variables of order at most 2.

Remark: Let Γ be a little quantified context, $T = (x_1 : U_1)...(x_p : U_p)(w\ s_1\ ...\ s_q)$ a little type in Γ whose all free existential variables are of order at most 2, and $k \in [0,p]$, then:

- $\Gamma' = \Gamma@[\exists x_1 : U_1; ...; \exists x_k : U_k]$ is a little quantified context,
- $T' = (x_{k+1} : U_{k+1})...(x_p : U_p)(w\ s_1\ ...\ s_q)$ is a little type in Γ',
- all the free existential variables of T' are of order at most 2.

Proof: By a simple induction on k.

Remark: Let Γ be a quantified context, T be a little type in Γ and σ be a well-typed substitution in Γ. Then $\sigma(T)$ is a little type in $\sigma(\Gamma)$.
Proof: By induction on T. Let $T = (y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$. For all i:

- either U_i is of order at most 2 in $\Gamma@[\exists y_1 : U_1; ...; \exists y_{i-1} : U_{i-1}]$ or y_i has no occurrence in $(y_{i+1} : U_{i+1})...(y_p : U_p)(w\ s_1\ ...\ s_q)$
- U_i is a little type in $\Gamma@[\exists y_1 : U_1; ...; \exists y_{i-1} : U_{i-1}]$.

Using the remark of preservation of types with well-typed substitutions:

- either $\sigma(U_i)$ is of order at most 2 in $\sigma(\Gamma)@[\exists y_1 : \sigma(U_1); ...; \exists y_{i-1} : \sigma(U_{i-1})]$ or y_i has no occurrence in $(y_{i+1} : \sigma(U_{i+1}))...(y_p : \sigma(U_p))\sigma((w\ s_1\ ...\ s_q))$

and using the induction hypothesis:

- $\sigma(U_i)$ is a little type in $\sigma(\Gamma)@[\exists y_1 : \sigma(U_1); ...; \exists y_{i-1} : \sigma(U_{i-1})]$.

Remark: If Γ is a little quantified context and σ a well-typed substitution in Γ then $\sigma(\Gamma)$ is a little quantified context.
Proof: By induction on the length of Γ.

4 A proof synthesis method

4.1 A pattern matching algorithm

If Γ is a quantified context, t is a term that has all its free existential variables of order at most 2 and t' a ground term then the matching problem $t = t'$ is decidable. An algorithm is given in [11].

Here we need a symmetrical algorithm, i.e. a unification function $\tilde{U}(\Gamma, t, t')$ that associates to each triple (Γ, t, t') where Γ a quantified context, and t and t' two terms in which free existential variables are of order at most 2 a set of substitutions well typed in Γ such that $\sigma(t) = \sigma(t')$. If one of the two terms t and t' is ground then the set $\tilde{U}(\Gamma, t, t')$ is a complete set of unifiers. If none of these terms is ground, the set $\tilde{U}(\Gamma, t, t')$ may for instance be empty, but we can imagine more powerful unification functions that gives some unifiers even if none of the terms is ground.

We give two properties of this algorithm that will be used in the following.

Remark: If one of the two terms t and t' is ground, in a substitution $\sigma \in \tilde{U}(\Gamma, t, t')$, an existential variable is not instanciated ($\sigma(x) = x$) or is instanciated by a ground term.

Definition: Let t be a term, we define $|t|$ the *size* of t by induction:
$|x| = 1$ if x is a variable,
$|Prop| = |Type| = |Kind| = 1$,
$|(u\ v)| = |u| + |v|$,
$|[x : T]u| = |u|$,
$|(x : T)u| = |u|$.

Definition: Let τ be a substitution $\tau = \{< x_1, t_1 >, ..., < x_p, t_p >\}$. We define $|\tau|$ the *size* of τ as the sum of the sizes of the t_i.

Definition: Let Γ be a context and t a term. t can be written in a unique way:

$$t = [x_1 : T_1]...[x_n : T_n](y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$$

t is said to *have no product* if $p = 0$ and $s_1, ..., s_q$ have no product.

Definition: A substitution $\tau = \{< x_1, t_1 >, ..., < x_p, t_p >\}$ is said to *have no product* if and only if all the t_i have no product.

Remark: Let σ be a substitution given by the algorithm (i.e. such that there exists Γ, t and t' such that $\sigma \in U(\Gamma, t, t')$) and ρ a substitution such that $\sigma \le \rho$ then there exists η such that $\eta \circ \sigma = \rho$ and $|\eta| \le |\rho|$. Moreover if ρ has no product, then η has also no product.

4.2 An inference system

We define the following inference system:

Rule 0: Erasement

Let $\Gamma = \Delta@[d]$ be a quantified context. If d is a constant definition or an universal variable declaration then we deduce the context Δ.

Rule 1: Resolution

Let $\Gamma = \Delta@[\exists x : P]$ be a quantified context. For every universal variable or constant declared in Δ, $t : T$, $T = (y_1 : U_1)...(y_p : U_p)V$, we let $\Xi = \Delta@[\exists y_1 : U_1, ..., \exists y_p : U_p]$. For every $\sigma \in \tilde{U}(\Xi, V, P)$ we deduce the context $\sigma(\Xi)$.

Rule 2: Introduction

Let $\Gamma = \Delta@[\exists x : P]$ be a quantified context. If $P = (y : U)T$ then we let z be a new variable, we deduce the context $\Gamma' = \Delta@[\forall y : U]@[\exists z : T]$.

Rule 3: *Prop*, *Type* and *Kind*

Let $\Gamma = \Delta@[\exists x : P]$ be a quantified context. For every
$\sigma \in \tilde{U}(\Delta, P, Prop) \cup \tilde{U}(\Delta, P, Type) \cup \tilde{U}(\Delta, P, Kind)$ we deduce the context $\sigma(\Delta)$.

Definition: Let Γ be a quantified context. A *list issued* of Γ is a list $[\Gamma_1; ...; \Gamma_n]$ of quantified contexts such that: $\Gamma_1 = \Gamma$ and Γ_{i+1} is deduced of Γ_i by one of the rules above.
A *derivation* of Γ is a list issued of Γ such that Γ_n is universally quantified.

4.3 Soundness

Remark: If Γ is a little quantified context, then all Γ' deduced of it by one of the rules above is a little quantified context.
Proof: By induction on the length of the derivation of Γ', we use the two following remarks:

- If Δ is a little quantified context and: $T = (x_1 : U_1)...(x_p : U_p)V$ a little type in Δ such that all the free existential variables of T are of order at most 2, then:

$$\Xi = \Delta@[\exists x_1 : U_1; ...; \exists x_p : U_p]$$

  is a little quantified context and V is a little type in Ξ in which all the free existential variables are of order at most 2.

- If Γ is a little quantified context and σ a well-typed substitution in Γ then $\sigma(\Gamma)$ is a little quantified context.

Remark: Soundness of the Erasement rule

Let Γ be a quantified context and Γ' a quantified context deduced of it with the Erasement rule then if we know a substitution τ' solution of Γ' we can construct a substitution τ solution of Γ.
Proof: We let $\tau = \tau'$, $\tau(\Gamma')$ is a universally quantified context then so is $\tau(\Gamma)$.

Remark: Soundness of the Resolution rule

Let Γ be a quantified context and Γ' a quantified context deduced of it by the Resolution rule, then if we know a substitution τ' solution of Γ' we can construct a substitution τ solution of Γ.
Proof: Let us write $\Gamma = \Delta @ [\exists x : P]$ and let $t : T = (y_1 : U_1)...(y_p : U_p)V$ be the universal variable or the constant of Δ and σ the substitution used in that deduction.
Let $\Xi = \Delta @ [\exists y_1 : U_1; ...; \exists y_p : U_p]$, $\Gamma' = \sigma(\Xi)$. The substitution $\rho = \tau' \circ \sigma$ is a solution of Ξ.

We first prove that: $\forall j \in [1,p] \;\; \rho(\Delta) \vdash \rho(y_j) : \rho(U_j)$.
ρ is a solution of Ξ, so $\forall j \in [1,p] \;\; \rho(\Xi_{|y_j}) \vdash \rho(y_j) : \rho(U_j)$.
Thus $\forall j \in [1,p] \;\; \rho(\Xi_{|y_j} @ [y_j : U_j]) = \rho(\Xi_{|y_j})$ i.e. $\forall j \in [1,p] \;\; \rho(\Xi_{|y_{j+1}}) = \rho(\Xi_{|y_j})$.
Therefore, by induction, $\forall j \in [1,p] \;\; \rho(\Xi_{|y_j}) = \rho(\Xi_{|y_1})$ i.e. $\forall j \in [1,p] \;\; \rho(\Xi_{|y_j}) = \rho(\Delta)$.
So $\forall j \in [1,p] \;\; \rho(\Delta) \vdash \rho(y_j) : \rho(U_j)$.

Then we prove that: $\rho(\Delta) \vdash \rho(t) : (x_1 : \rho(U_1))...(x_p : \rho(U_p))\rho(T)$.
$\Delta \vdash t : (x_1 : U_1)...(x_p : U_p)T$, so $\rho(\Delta) \vdash \rho(t) : (x_1 : \rho(U_1))...(x_p : \rho(U_p))\rho(T)$.

We deduce: $\rho(\Delta) \vdash (\rho(t) \; \rho(y_1) \; ... \; \rho(y_p)) : \rho(T)$.

At last we prove: $\rho(T) = \rho(P)$.
$\sigma(T) = \sigma(P)$, so $\tau'(\sigma(T)) = \tau'(\sigma(P))$, i.e. $\rho(T) = \rho(P)$.

Then we conclude: $\rho(\Delta) \vdash (\rho(t) \; \rho(y_1) \; ... \; \rho(y_p)) : \rho(P)$.

Let $\tau(x) = (\rho(t) \; \rho(y_1) \; ... \; \rho(y_p))$ and $\tau(u) = \rho(u)$ for $u \neq x$.
Obviously, for all existential variable $x : P$ of Γ, $\tau(\Gamma_{|x}) \vdash \tau(x) : \tau(P)$. Thus τ is a solution of Γ.

Remark: Soundness of the Introduction rule
Let Γ be a quantified context and Γ' a context deduced from it by the Introduction rule. If we know a substitution τ' solution of Γ' then we can construct a substitution τ solution of Γ.
Proof: Let us write $\Gamma = \Delta @ [\exists x : P]$ and $P = (y : U)T$, and let $\Gamma' = \Delta @ [\forall y : U] @ [\exists z : T]$.
The substitution τ' is a solution of Γ', so for all existential variable $x : P$ of Δ, $\tau'(\Gamma_{|x}) \vdash \tau'(x) : \tau'(P)$ and $\tau'(\Delta @ [\forall y : U]) \vdash \tau'(z) : \tau'(P)$.

Let $\tau(x) = [y : U]\tau'(z)$ and $\tau(u) = \tau'(u)$ for $u \neq x$.
Obviously, for all existential variable x of Γ, $\tau(\Gamma_{|x}) \vdash \tau(x) : \tau(P)$. Thus τ is a solution of Γ.

Remark: Soundness of the rule *Prop*, *Type* and *Kind*
Let Γ be a quantified context and Γ' a context deduced by the rule $Prop$, $Type$ and $Kind$, if we know a substitution τ' solution of Γ' we can construct a substitution τ solution of Γ.
Proof: Let us write: $\Gamma = \Delta @ [\exists x : P]$ and let σ be the substitution used in that deduction.
$\Gamma' = \sigma(\Delta)$. The substitution $\rho = \tau' \circ \sigma$ is a solution of Δ.

If $\sigma(P) \in \{Prop, Type\}$ then let $t = (x : \sigma(P))x$ and if $\sigma(P) = Kind$ then let $t = Type$.
Let $\tau(x) = t$ and $\tau(u) = \rho(u)$ for $u \neq x$.
Obviously, for all $x : P$ existential variable of Γ, $\tau(\Gamma_{|x}) \vdash \tau(x) : \tau(P)$. Thus τ is a solution of Γ.

Remark: Soundness of the method
Let Γ be a quantified context. If we know a derivation of Γ then we can construct a substitution τ solution of Γ.
Proof: Let $[\Gamma_1; ...; \Gamma_n]$ be a derivation of Γ. $\Gamma_1 = \Gamma$ and Γ_n is universally quantified. We build, by decreasing induction p, a sequence of substitutions $(\tau_p)_p$ such that for all p, τ_p is a solution of Γ_p.

For $p = n$ let $\tau_n = Id$. N_n is universally quantified, thus so is $Id(N_n)$. Then let us assume that the substitution τ_{p+1} is constructed, we have seen in the previous remarks how to construct the substitution τ_p knowing τ_{p+1} and the rule which permits to deduce N_{p+1} from N_p.

So we can construct a substitution $\tau = \tau_1$ solution of Γ.

Remark: Let Γ be a little (non quantified) context, P a term of type $Prop$, $Type$ or $Kind$ in Γ, which is a little type, x a variable. If we know a derivation $[\Gamma_1; ...; \Gamma_n]$ of $\Gamma@[\exists x : P]$ then we can construct a proof t of P in Γ. t is called the proof denoted by the derivation $[\Gamma_1; ...; \Gamma_n]$.

Proof: From the previous remark, we can construct a substitution τ such that $\tau(\Gamma) \vdash \tau(x) : \tau(P)$.

Since Γ is universally quantified and P is well formed in Γ, $\tau(\Gamma) = \Gamma$ and $\tau(P) = P$. The term $t = \tau(x)$ is such that $\Gamma \vdash t : P$.

4.4 Incompleteness

Even if $\tilde{U}$ is a complete unification procedure, this method is incomplete.

4.4.1 Examples of non synthesized proof

Example 1: Induction loading

There is no deduction for proofs that need the application of a term to a proposition, a type or a predicate that has an initial product, $[x_1 : T_1]...[x_n : T_n](y : U)V$, when this product is then eliminated. In particular proofs using induction loading cannot be synthesized.

For instance:

$$\dfrac{\dfrac{\dfrac{(P : Prop)(I\ P) \to P \qquad (A \to B) : Prop}{I\ (A \to B) \to (A \to B)} \qquad I(A \to B)}{(A \to B)} \qquad A}{B}$$

With the goal B and the universal variable: $\{(P : Prop)(I\ P) \to P, I(A \to B), A\}$, the only rule that can be used is the Resolution rule with the proposition $(P : Prop)(I\ P) \to P$ and the substitution $\{< P, B >\}$. The generated goal is $(I\ B)$ from which we can only generate: $(I\ (I\ B))$, etc. The system "does not see" that $u : (P : Prop)(I\ P) \to P$ must be applied to $A \to B$

Example 2:

$$\dfrac{\dfrac{(P : T \to Prop)((P\ t) \to A) \qquad [x : T]((B\ x) \to (C\ x)) : T \to Prop}{((B\ t) \to (C\ t)) \to A} \qquad (B\ t) \to (C\ t)}{A}$$

Here, with the goal A the only rule that can be used is the Resolution with the proposition $(P : T \to Prop)((P\ t) \to A)$ this generates two goals: $P : T \to Prop$ and $(P\ t)$ with this goal the only rule that can be used is the Resolution with $(B\ t) \to (C\ t)$, this generates the goal $(B\ t)$ that cannot be proved. The system "does not see" that $(P : T \to Prop)((P\ t) \to A)$ must be applied to $[x : T]((B\ x) \to (C\ x))$.

Example 3: Third order proofs

If $u : (w : Prop \rightarrow Prop)(w\ (x : A)B)$ and $v : A$ then $(u\ [x : Prop]x\ v) : B$. This proof cannot be synthesized because with the goal B, no rule is applicable.

4.4.2 A set of synthesized proofs

Remark: Let Γ be a little quantified context, $x : T$ an existential variable of Γ and t, u little terms with no initial products. We assume that either T is of order at most 2 or x has no occurrence in t. Then $a = t[x \leftarrow u]$ normalizes to a little term with no initial product.
Proof: For a term a we write $len(a)$ the maximum length of a reduction issued of a. We prove the property by induction $len(a)$.
Since t has no initial product:

$$t = [y_1 : T_1]...[y_p : T_p](w\ s_1\ ...\ s_q)$$

$$a = [y_1 : T_1[x \leftarrow u]]...[y_p : T_p[x \leftarrow u]](w[x \leftarrow u]\ s_1[x \leftarrow u]\ ...\ s_q[x \leftarrow u])$$

Let:

$$b = (w[x \leftarrow u]\ s_1[x \leftarrow u]\ ...\ s_q[x \leftarrow u]))$$

We want to prove that the normal form of this term has no initial product.

- If $w \neq x$ then:
$$b = (w\ s_1[x \leftarrow u]\ ...\ s_q[x \rightarrow u])$$
which normalizes to an application i.e. a term with no initial product.

- If $w = x$ then:
$$b = (u\ s_1[x \leftarrow u]\ ...\ s_q[x \leftarrow u])$$
Let us write s'_i the normal form of $s_i[x \leftarrow u]$. The type of $x = w$ is of order at most 2 so the types of the s_i are first order, thus so are the types of the s'_i, so the s'_i have no initial products.
The term b reduces to $b' = (u\ s'_1\ ...\ s'_q)$. Using the induction hypothesis and with a simple induction on q we prove that b' normalizes to a term with no initial product.

Thus a normalizes to a term with no initial product.

Remark: Let Γ be a context and t and t' two little terms in Γ such that t' has no initial product and $(t\ t')$ is well-typed. Then the type of t have one initial product more than the type of $(t\ t')$.
Proof: Let T be the type of the term t. Since T is of type $Prop$, $Type$ or $Kind$ it can be written:

$$T = (y_1 : U_1)(y_2 : U_2)...(y_p : U_p)(w\ s_1\ ...\ s_q)$$

Since the term t can be applied to the term t', $p \geq 1$.
The type of $(t\ t')$ is:

$$(y_2 : U_2[y_1 \leftarrow t'])...(y_p : U_p[y_1 \leftarrow t'])(w\ s_1\ ...\ s_q)[y_1 \leftarrow t']$$

Let $\Gamma' = \Gamma@[\exists y_2 : U_2[y_1 \leftarrow t']; ...; \exists y_p : U_p[y_1 \leftarrow t']; \exists y_1 : U_1]$.
The terms $(w\ s_1\ ...\ s_q)$ and t' are well formed in Γ', they are both little terms with no initial product

and either U_1 is of order at most 2 in Γ' or y_1 has no occurrence in $(w\ s_1\ ...\ s_q)$.
Using the previous remark $(w\ s_1\ ...\ s_q)[y_1 \leftarrow t']$ normalizes on a term with no product, so the type of t has p initial products and the type of $(t\ t')$ $p-1$.

Remark: Let Γ be a context and $v = (w\ s_1\ ...\ s_q)$ a little term well formed in Γ such that $s_1, ..., s_q$ have no initial product[7] then the type of w has at least q initial products, i.e. is of the form $(y_1 : T_1)...(y_q : T_q)T$.
Proof: The term $(w\ s_1\ ...\ s_{q-1})$ can be applied to a term. So, its type has at least one initial product. We then prove by induction, and using the previous remark that since s_{i+1} has no product the type of $(w\ s_1\ ...\ s_i)$ has at least $q-i$ initial products.

Remark: Let Γ be a quantified context and $x_1, ..., x_n$ be the existential variables of Γ (we assume that they appear in this order in Γ) and $\tau = \{< x_1, t_1 >, ..., < x_n, t_n >\}$ a substitution with no product, solution of Γ. Then there exists a derivation of Γ.
Proof: By induction on the size of τ.

Let $P_1, ..., P_n$ be the types of x_i, $t = \tau(x_n)$ and $P = \tau(P_n)$, by hypothesis $\tau(\Gamma_{|x_n}) \vdash t : P$ and t has no product. It can therefore be written:

$$t = [y_1 : T_1]...[y_p : T_p](w\ s_1\ ...\ s_q)$$

where w is a variable or one of the symbols $Prop$ or $Type$ but not the symbol $Kind$.
Let $v = (w\ s_1\ ...\ s_q)$ and Q be the type of v.

$$P = (y_1 : T_1)...(y_p : T_p)Q$$

Let us write $P_n = (y_1 : T'_1)...(y_{p'} : T'_{p'})Q'$.
Since τ has no product and Q' is a little term with no initial product, $\tau(Q')$ has no initial product, so $p = p'$, $\forall i \in [1, p] T_i = \tau(T'_i)$ and $Q = \tau(Q')$.
We build a sequence of contexts, by erasing all the universal variables and constant declared after x_n, then by introducing $T'_1, ..., T'_p$.
We deduce the context:

$$\Gamma' = \Gamma_{|x_n} @ [\forall y_1 : T'_1; ...; \forall y_p : T'_p] @ [\exists x : Q']$$

- If $w = Prop$ or $w = Type$ then $Q = Type$ or $Q = Kind$ so $\tau(Q') = Type$ or $Kind$.
  Let $\rho = \tau - \{< x, \tau(x) >\}$. $|\rho| < |\tau|$ and $\rho(Q') = Type$ or $Kind$. Let σ be the substitution of $\tilde{U}(\Gamma'_{|x}, Q', Type) \cup \tilde{U}(\Gamma'_{|x}, Q', Kind)$ less or equal than ρ and η a substitution with no product such that $\rho = \eta \circ \sigma$ and $|\eta| \leq |\rho|$.
  We deduce the context $\sigma(\Gamma_{|x_n} @ [\forall y_1 : T'_1; ...; \forall y_p : T'_p])$ with the rule $Prop$, $Type$ and $Kind$ and the substitution σ. The substitution η is a solution of this context and $|\eta| \leq |\rho| < |\tau|$, so by induction hypothesis there exists a derivation of this context, so there exists also a derivation of Γ.

- If w is a variable, the term t has no product, neither has v. So using the previous remark, the type of w has at least q initial products, let $(z_1 : V_1)...(z_q : V_q)V$ be this type. Let W_i be the type of s_i, $\Xi = \Gamma_{|x_n} @ [\forall y_1 : T'_1; ...; \forall y_p : T'_p] @ [\exists z_1 : V_1; ...; \exists z_q : V_q]$ and
  $\rho = \tau - \{< x_n, \tau(x_n) >\} \cup \{< z_1, s_1 >, ..., < z_q, s_q >\}$. ρ is a solution of Ξ and $|\rho| < |\tau|$.

[7] actually, it is enough to assume that $s_1, ..., s_{q-1}$ have no initial product.

The term t and the substitution τ have no product, then neither has the substitution ρ. Let σ be the element of $\tilde{U}(\Xi, V, Q')$ less or equal to ρ and η a substitution with no product such that $\rho = \eta \circ \sigma$ and $|\eta| \leq |\rho|$.
We deduce the context $\sigma(\Xi)$ with a Resolution rule with the symbol w and the substitution σ. The substitution η is a solution of $\sigma(\Xi)$ and $|\eta| \leq |\rho| < |\tau|$, so by induction hypothesis there exists a derivation of this context, so there exist also a derivation of Γ.

Remark: Let Γ be a little (non quantified) context and P a proposition or a type which is a little type and which has a proof t with no product. There exists a derivation of $\Gamma@[\exists x : P]$.
Proof: Let $\tau = \{< x, t >\}$. Since t has no product, neither has τ. τ is a solution of $\Gamma@[\exists x : P]$ so there exists a derivation of this context.

N.B.: Let P be a proposition or a type and $t = [x_1 : T_1]...[x_n : T_n](y_1 : U_1)...(y_p : U_p)(w\ s_1\ ...\ s_q)$ a proof of P. If $p \neq 0$, P is equal to $(x_1 : T_1)...(x_p : T_p)K$ with $K \in \{Prop, Type, Kind\}$. The rule $Prop$, $Type$ and $Kind$ gives a proof for these propositions, so it is the products in the s_i that restrict completeness.

N.B.: We have seen that a proposition that has a proof with no product has a derivation. The converse is false, indeed, the proof:

$$\cfrac{\cfrac{(P : Prop)(I\ P) \to P \qquad (A \to B) : Prop}{I\ (A \to B) \to (A \to B)} \qquad I(A \to B)}{(A \to B)}$$

has a derivation: the goal $(A \to B)$ matches the head of $(P : Prop)(I\ P) \to P$, this generates the goal $I(A \to B)$ that is in the context and the synthesized proof is: $t = (u\ (A \to B)\ v)$ if $u : (P : Prop)(I\ P) \to P$ and $v : I(A \to B)$.

4.5 Premises

As we have seen in the informal presentation of the method we need, in order to avoid explosion of search time when the context grows, to indicate a subset π of Γ (the premises) that are the propositions that can be used to solve the goals.

Definition: A proof synthesis problem is a couple (Γ, π) where Γ is a little quantified context and π a subset of Γ.

Definition: Un problem (Γ, π) is said to be *universally quantified* if Γ is universally quantified.

Definition: Let (Γ, π) be a proof synthesis problem and σ a substitution well-typed in Γ. We define the problem $\sigma(\Gamma, \pi)$ by induction on the length of Γ:

- If $\Gamma = []$ then $\pi = \emptyset$, We let $\sigma(\Gamma, \pi) = ([], \emptyset)$.
- If $\Gamma = \Delta@[x = t : T]$ then let $(\Gamma', \pi') = \sigma(\Delta, \pi - \{x\})$ and $\pi'' =$ if $x \in \pi$ then $\pi' \cup \{x\}$ else π'. We let $\sigma(\Gamma, \pi) = (\Gamma'@[x = \sigma(t) : \sigma(T)], \pi'')$.
- If $\Gamma = \Delta@[\forall x : T]$ then let $(\Gamma', \pi') = \sigma(\Delta, \pi - \{x\})$ and $\pi'' =$ if $x \in \pi$ then $\pi' \cup \{x\}$ else π'. We let $\sigma(\Gamma, \pi) = (\Gamma'@[\forall x : \sigma(T)], \pi'')$.

- If $\Gamma = \Delta@[\exists x : T]$ then let $\tau = \sigma - \{< x, \sigma(x) >\}$ and $(\Gamma', \pi') = \tau(\Delta, \pi - \{x\})$.
  Let $y_1 : U_1, ..., y_k : U_k$ be the variables of $\sigma(x)$ that are not declared in Γ'.
  We let $\sigma(\Gamma, \pi) = (\Gamma'@[\exists y_1 : U_1; ...; \exists y_k : U_k], \pi')$

Remark: Let (Γ, π) be a problem and σ and τ two substitutions then $\sigma \circ \tau(\Gamma, \pi) = \sigma(\tau(\Gamma, \pi))$
Proof: By induction on the length of Γ.

We modify the inference system, in order first to indicate the set π associated to each context and then to restrict the Resolution rule.

- In the Erasement rule we have $(\Gamma = \Delta@[d], \pi)$ and we deduce $(\Delta, \pi - \{d\})$.
- In the Resolution rule, we have $(\Gamma = \Delta@[\exists x : P], \pi)$. We can only use a constant or a universal variable $t : T$ if it is an element of π. When the premise used is $t : T$ with $T = (y_1 : U_1)...(y_p : U_p)V$, we let $\Xi = \Delta@[\exists y_1 : U_1, ..., \exists y_p : U_p]$, for every $\sigma \in \tilde{U}(\Xi, V, P)$ we deduce $\sigma(\Xi, \pi)$.
- In the Introduction rule we have $(\Gamma = \Delta@[\exists x : P], \pi)$ with $P = (y : U)T$. We let z be a new variable, $\Gamma' = \Delta@[\forall y : U]@[\exists z : T]$, we deduce the problem: $(\Gamma', \pi \cup \{y\})$.
- In the rule *Prop*, *Type* and *Kind* we have $(\Gamma = \Delta@[\exists x : P], \pi)$.
  For every $\sigma \in \tilde{U}(\Delta, P, Prop) \cup \tilde{U}(\Delta, P, Type) \cup \tilde{U}(\Delta, P, Kind)$ we deduce $\sigma(\Delta, \pi)$.

Remark :
The completeness remark becomes : if Γ is a little context and P is a proposition or a type which is a little type and which has a proof t with no product then there exist a set π such that there exist a derivation of $(\Gamma@[\exists x : P], \pi)$.

The set π which contain all the constants and universal variables of Γ obviously works, but for efficiency, it is better to give a set as small as possible.

4.6 Algorithms

4.6.1 A semi-algorithm: Resolution trees

Let Γ be a quantified context, we build a tree which nodes are quantified contexts. The root is Γ and the sons of a context are all the contexts deduced by the rules of the system presented above. The branches of the tree are the lists issued of Γ. The empty nodes are success nodes, they are in bijection with the derivations of Γ. The non empty nodes that have no sons are failure nodes.

A semi-algorithm of proof synthesis is to enumerate the nodes of the tree in order to find a success node. This semi-algorithm has two causes of incompleteness: first the restriction on unification and then the intrinsic restriction of top-down Resolution.

In order to overcome the first cause of incompleteness, we would have to take for $\tilde{U}$ a complete unification procedure (an extension of Jensen-Pietrzykowski's algorithm [21] to the Calculus of Constructions). Since the tree of this algorithm is not finitely branching, we would need two levels of inter-leaving to enumerate the nodes of the Resolution tree. In order to have only one level of inter-leaving, we could restrict unification to preunification (an extension of Huet's algorithm [18] to the Calculus of Constructions) and use a constraint propagation method in the Resolution tree [19]. In order to overcome the second cause of incompleteness we could, in the Calculus restricted to little terms, add a rule like the Splitting rule of the system presented in [19].

4.6.2 An always terminating algorithm

In order to get rid of the infinite branches of the Resolution tree, we can bound the depth of this tree. More precisely, we give at the beginning an index to the goal to be proved and when we apply a Resolution rule to a goal of index n we give to the new goals the index $n-1$. When we apply an Introduction rule, we keep the same index to the new goal. It is only possible to apply a rule to a goal whose index is strictly positive.

When we use such an algorithm of proof synthesis in a Vernacular, this restriction means that the user has to break up his proofs in sufficiently small pieces.

So this algorithm has three causes of incompleteness: the intrinsic incompleteness of top-down Resolution, the restriction on the unification and the bound on the Resolution tree.

5 An allusive Vernacular

In this section we are going to use this algorithm to design a complete Vernacular for little terms.

5.1 Proofs decomposition

We want to use this proof synthesis algorithm to write mathematical texts where demonstrations are sequences of lemmas, each lemma having a proof synthesized with previous lemmas as premises. The three restrictions of the proof synthesis method lead to three kinds of decomposition of proofs:

- The products in the proofs:

  In the example:

$$\dfrac{\dfrac{\dfrac{(P:Prop)(I\ P)\to P \qquad (A\to B):Prop}{I\ (A\to B)\to(A\to B)} \qquad I(A\to B)}{(A\to B)} \qquad A}{B}$$

  We have to prove first the proposition $A\to B$ and then the proposition B using the premise $A\to B$. In order to prove the first proposition we must match directly $A\to B$ with P. This is the reason why it is important not to introduce always the hypothesis A, but also to try to prove directly the proposition $A\to B$.

- Unification is restricted to pattern matching:

  This is not a problem while goals are ground. When we generate a goal with an existential variable we have to put among the premises a ground proposition that matches the goal. We have to prove this premise before the proposition.

- The proof tree is bounded:

  We have to break up the proof in sufficiently small pieces.

5.2 An inference system

We write $\Gamma \leadsto P$ the assertion that there exists a set of premises π and a derivation of $(\Gamma@[\exists x : P], \pi)$ and we consider assertions $\Gamma \rhd P$ meaning intuitively that there exists a text in Vernacular which is a demonstration of P.

We want, a priori, to have only one rule that allows to synthesize the proof of a new proposition, using already proved ones. Actually we need also another rule, which allows to introduce explicitly an hypothesis or a variable. Indeed let us imagine that we want to prove a proposition $A \to B$ in introducing the hypothesis A then proving B. If a proof of B cannot be automatically synthesized and for instance we have to prove a lemma C (using the hypothesis A) before, we cannot let the system introduce automatically the hypothesis A, we have to do it by hand.

Rule 1: Synthesis

$$\frac{\Gamma \rhd Q_1 \quad ... \quad \Gamma \rhd Q_n \qquad \Gamma@[c_1 : Q_1; ...; c_n : Q_n] \leadsto P}{\Gamma \rhd P}$$

If the c_i have no occurrence in P.

Rule 2: Explicit introduction

$$\frac{\Gamma@[x : Q] \rhd P}{\Gamma \rhd (x : Q)P}$$

5.3 Soundness and completeness

Remark: Soundness

If we know a derivation of $\Gamma \rhd P$ then we can construct a proof t of P in Γ. t is called the proof denoted by the derivation.

Proof: By induction on the length of the derivation of $\Gamma \rhd P$.

If the last rule of the derivation is the rule *Synthesis* we have by the soundness of the proof synthesis method a term u such that:

$$\Gamma@[c_1 : Q_1, ..., c_n : Q_n] \vdash u : P$$

and by induction hypothesis terms $v_1, ..., v_n$ such that:

$$\Gamma \vdash v_i : Q_i$$

so:

$$\Gamma \vdash u[c_1 \leftarrow v_1, ..., c_n \leftarrow v_n] : P$$

If the last rule of the derivation is the rule *Explicit introduction* then $P = (x : Q)R$ and we have by induction hypothesis a term u such that:

$$\Gamma@[x : Q] \vdash u : R$$

$$\Gamma \vdash [x : Q]u : P$$

Remark: If $x : P$ is a variable declared in Γ then $\Gamma \leadsto P$.
Proof: Let $\pi = \{x : P\}$. We derive universally quantified problem from $(\Gamma@[\exists x : P], \pi)$ by the Resolution rule with $\sigma = Id$.

Remark: $\Gamma \leadsto Prop$, $\Gamma \leadsto Type$ and $\Gamma \leadsto Kind$
Proof: Let $K \in \{Prop, Type, Kind\}$, we derive a universally quantified problem from $(\Gamma@[\exists x : K], \emptyset)$ with the rule *Prop*, *Type* and *Kind*.

Remark: Completeness
Let Γ be a context, P a little type and t a term such that $\Gamma \vdash t : P$, then $\Gamma \ \triangleright P$.
Proof: By induction on the size of t.

- If t is a product, $t = (x : Q)u$ then $P \in \{Prop, Type, Kind\}$. Using the previous remark $\Gamma \leadsto P$, so using the rule *Synthesis* $\Gamma \ \triangleright P$.
- If $t = Prop$ or $t = Type$ (t cannot be equal to $Kind$) then $P = Type$ or $P = Kind$. Using a previous remark, $\Gamma \leadsto P$ so using the rule *Synthesis* $\Gamma \ \triangleright P$.
- If t is a variable x of Γ then $\Gamma \leadsto P$ so using the rule *Synthesis*: $\Gamma \ \triangleright P$.
- If t is an abstraction we write: $t = [x : Q]u$, let R be the type of u in $\Gamma@[x : Q]$.
  By induction hypothesis, $\Gamma@[x : Q] \ \triangleright R$, using the rule *Explicit introduction*, $\Gamma \ \triangleright P$
- If t is an application, then we write $t = (u \ v)$. Let $\tau = \{< y, v >\}$ and R be the type of u. The term u can be applied, so its type R is ofthe form $(y : Q)S$.
  $u : (y : Q)S$, so $(u \ v) : S[y \leftarrow v]$ thus $P = S[y \leftarrow v]$ i.e. $\tau(S) = P$.
  Let σ be the element of $\tilde{U}(\Gamma@[\exists y : Q], P, S)$ which is less or equal of τ. The domain of σ is a subset of the singleton $\{y\}$, and y is unbound or bound to a ground term.
  - If y is bound to a ground term, then this term is less or equal than v and therefore it can only be v. So the substitution $\tau = \{< y, v >\}$ is an element of $\tilde{U}(\Gamma@[y : Q], P, S)$, thus $\Gamma@[c : (y : Q)S] \leadsto P$, indeed let $\Gamma' = \Gamma@[c : (y : Q)S]$ and $\pi = \{c : (y : Q)S\}$, from $(\Gamma'@[\exists x : P], \pi)$ we can deduce by the Resolution rule with the premise c and the substitution τ the problem (Γ', π) which is a universally quantified problem.
    Moreover $\Gamma \vdash u : (y : Q)S$, so by induction hypothesis $\Gamma \ \triangleright (y : Q)S$. So using the rule *Synthesis* $\Gamma \ \triangleright S$.
  - If y is unbound by σ, then σ is the identity so $S = P$.
    $\Gamma@[c : (y : Q)P; d : Q] \leadsto P$, indeed let $\Gamma' = \Gamma@[c : (y : Q)P; d : Q]$ and $\pi = \{c : (y : Q)P, d : Q\}$, from $(\Gamma'@[\exists x : P], \pi)$ we can deduce by the Resolution rule with the premise c the substitution Id the problem $(\Gamma'@[\exists y : Q], \pi)$, then by the Resolution rule with the premise d and the substitution Id we deduce (Γ', π) which is a universally quantified problem.
    Moreover $\Gamma \vdash u : (y : Q)S$ and $\Gamma \vdash v : Q$, so by induction hypothesis $\Gamma \ \triangleright (y : Q)S$, $\Gamma \ \triangleright Q$. So using the rule *Synthesis*, $\Gamma \ \triangleright P$.

5.4 Length of derivations

With a better use of the proof synthesis method, we can write shorter proofs in this system, that will lead to shorter texts in Vernacular.

5.4.1 Using several lemmas together

When we have a proof $t = (u\ v_1\ ...\ v_n)$ where $v_1, ..., v_{n-1}$ have no initial product, we can write this proof in only one step.

Let P be the type of $(u\ v_1\ ...\ v_n)$, T_i the types of v_i and $(x_1 : Q_1)...(x_n : Q_n)S$ the type of u. Let ρ be the substitution $\{< x_1, v_1 >, ..., < x_n, v_n >\}$, ρ is such that $\rho(S) = P$ and $\rho(Q_i) = T_i$. Let σ be the element of $\tilde{U}(\Gamma@[\exists x_1 : Q_1; ...; \exists x_n : Q_n], (S, Q_1, ..., Q_n), (P, T_1, ..., T_n))$[8] less or equal than ρ. In σ every variable x_i is bound to a ground term or unbound. We separate the x_i in the ones which are bound to a ground term: the $x_{\alpha(i)}$ (this term can only be $v_{\alpha(i)}$) and the ones which are unbound: the $x_{\beta(i)}$.

$$\sigma = \{< x_{\alpha(1)}, v_{\alpha(1)} >, ..., < x_{\alpha(p)}, v_{\alpha(p)} >\}$$

Then:

$$\Gamma@[c : (x_1 : Q_1)...(x_n : Q_n)S; d_1 : T_{\beta(1)}; ...; d_q : T_{\beta(q)}] \rightsquigarrow P$$

Indeed, let:

$$\pi = \{c : (x_1 : Q_1)...(x_n : Q_n)S, d_1 : T_{\beta(1)}, ..., d_q : T_{\beta(q)}\}$$

From the problem:

$$(\Gamma@[\exists y : P], \pi)$$

we deduce by the Resolution rule with the premise c and the substitution σ the problem:

$$(\Gamma@[\exists x_1 : \sigma(Q_{\beta(1)}); ...; \exists x_q : \sigma(Q_{\beta(q)})], \pi)$$

Then by q Resolutions we solve x_i with the premise d_i, we deduce:

$$(\Gamma, \pi)$$

Moreover $\Gamma \vdash u : (x_1 : Q_1)...(x_n : Q_n)S$ and $\Gamma \vdash s_{\beta(i)} : T_{\beta(i)}$, so by induction hypothesis:

$$\Gamma\ \rhd (x_1 : Q_1)...(x_n : Q_n)S$$

$$\Gamma\ \rhd T_{\beta(i)}$$

So using the rule *Synthesis* we get:

$$\Gamma\ \rhd P$$

5.4.2 Iteration of the search for ground goals

When a generated goal is ground, instead of writing it as a lemma of the demonstration, we can add the premises which allow to prove that goal to the premises of the theorem and let the system look for a proof of it. Of course this must not be iterated a number of times greater than the bound on the deepness of the Resolution tree.

[8] here we need a simultaneous unification function, we can define as
$\sigma \in \tilde{U}(\Gamma@[\exists x_1 : Q_1; ...; \exists x_n : Q_n], (S, Q_1, ..., Q_n), (P, T_1, ..., T_n))$
if there exist a sequence σ_i such that
$\sigma_1 \in \tilde{U}(\Gamma@[\exists x_1 : Q_1; ...; \exists x_n : Q_n], S, P)$
$\sigma_2 \in \tilde{U}(\Gamma@[\exists x_1 : Q_1, ..., \exists x_n : Q_n], \sigma_1(Q_1), \sigma_1(T_1))$, etc.
and $\sigma = \sigma_{n+1} \circ ... \circ \sigma_1$.

5.4.3 Elimination of the products

Some propositions P are such that if they have a proof u that is applied to a term t that have an initial product, there exists another proof of the type of $(u\ t)$ where t has no initial product. We call these propositions *permutable propositions.* For instance, if we apply $P \wedge Q$ to $(x : A)B$ this proof can be re-written in another proof in which it is applied to B.

Remark: If A and B are two propositions, then $u : (A \wedge B) = (C : Prop)(A \to B \to C) \to C$, is permutable.
Proof: The proof:

$$\frac{(C : Prop)(A \to B \to C) \to C \qquad (x : P)Q : Prop}{(A \to B \to (x : P)Q) \to (x : P)Q}$$

can be re-written in:

$$\cfrac{\cfrac{\cfrac{(C : Prop)(A \to B \to C) \to C \qquad Q : Prop}{(A \to B \to Q) \to Q} \qquad \cfrac{\cfrac{[A \to B \to (x : P)Q] \quad [A] \quad [B] \quad [x : P]}{Q}}{A \to B \to Q}}{Q}}{(A \to B \to (x : P)Q) \to (x : P)Q}$$

Remark: It is easy to check that the disjunction, the existential and the absurd are permutable, the equality and the inductions on inductive types are not.

If we get rid that way of products, we get shorter proofs leading to shorter texts because we can use more lemmas in one rule.

5.4.4 Automatic introduction of hypothesis

When a proof is an abstraction $[x : T]u$ and a proof of the type of u is automatically synthesized then we must not introduce the hypothesis T by hand, we must let the system introduce the hypothesis automatically.

5.5 An allusive Vernacular

In the Elementary Vernacular [10] there is an instruction `Proof t.` where t is a term, which operational semantics was the following: when the proof checker meets the instruction `Proof t.` in a context Γ, it computes the type of t in Γ, eliminates the local elements declared since the last instruction `Statement`, checks that the type obtained that way is the same as the one given in the last instruction `Statement` (if it is not then it fails), then eliminates the local elements declared since the last instruction `Theorem` and adds this new theorem to the context.

We modify this Vernacular in replacing this instruction by `Using s1, ..., sn.` where $s1, ..., sn$ are symbolic names. The semantics of this new instruction is the following: when the proof checker meets the instruction `Using s1, ..., sn.` in a context Γ, it computes, using the type given in the last instruction `Statement` and the local elements declared since this last instruction, the goal to be proved then looks for a proof of this goal using the premises $\{s1, ..., sn\}$, fails if it does not find one, then eliminates the local elements declared since the last instruction `Theorem` and adds this new theorem to the context.

Remark: Completeness

For all inhabited proposition P, there exists a text in Vernacular which denotes a proof of this proposition.

Proof: We construct such a text from a derivation of $\Gamma \rhd P$.

- If the last rule used is the rule *Synthesis*, then there exists, by induction hypothesis, texts that define symbols $c_1, ..., c_n$ proof of $Q_1, ..., Q_n$. Let `<text>` be their concatenation.

  There exists also a set of premises `<premises>` of $\Gamma@[c_1 : Q_1, ..., c_n : Q_n]$ used in the synthesis of a proof of P. We build the text:

  ```
  Remark <name>.
  Statement P.
     <text>
  Using <premises>.
  ```

- If the last rule is the *Explicit introduction* then $P = (x : T)Q$. By induction hypothesis Q has a proof in Vernacular in the context $\Gamma@[x : T]$ that uses a set `<premises>` of premises (x may belong to `<premises>`). Let a text in Vernacular that proves Q in this context:

  ```
  Remark <name>.
     <text 1>
  Statement <stat>.
     <text 2>
  Using <premises>.
  ```

  We transform this text in:

  ```
  Remark <name>.
     Variable/Hypothesis x:T.
     <text 1>
  Statement <stat>.
     <text 2>
  Using <premises>.
  ```

Conclusion

In this paper we have formalized top-down Resolution as rewrite rules over quantified contexts. This presentation allows to define a restriction of this method which is always terminating and whose transitive closure is complete for a second order restriction to the Calculus of Constructions.

Using this method we have designed a Vernacular in which we do not need to write explicitly proof terms, we do not need to decompose the proofs in too small steps, correctness is decidable and which is complete for the second order restriction to the Calculus of Construction.

An improvement would be to design a more powerful theorem prover to make the Vernacular more allusive, for instance we could use a unification function which also is complete for first order problems, and iterate the search of proofs for some generated goals. Another improvement could concern the way we indicate the premises that can be used by the theorem prover, for instance we could avoid writing a premise if it is the last (or one of the lasts) proposition proved.

At last, let us remark that this Vernacular only works for a restriction of the Calculus of Constructions. We have then to generalize these algorithms to higher order or to prove that this restriction really makes sense.

References

[1] H. Barendregt. "Introduction to Generalised Type Systems". To appear in "Proceedings of the third Italian Conference on Theoretical Computer Science". Ed. U. Moscati, Mantova, 2-4 November 1989, World Scientific Publishing Co., Singapore.

[2] N.G. de Bruijn. "The Mathematical Vernacular, A Language For Mathematics With Typed Sets." Proceedings of the Workshop on Programming Logic, Marstrand, Sweden, 1987.

[3] N.G. de Bruijn. "The Mathematical Vernacular: Examples." Unpublished manuscript.

[4] Th. Coquand. "Une Théorie des Constructions." Thèse de troisième cycle, Université Paris VII, 1985.

[5] Th. Coquand. "An analysis of Girard's paradox." First IEEE Symposium on Logic in Computer Science, Boston, June 1986, 227-236.

[6] Th. Coquand. "Metamathematical investigations of a Calculus of Constructions." To appear, proceedings of the Talin Conference, 1990. Also in [9].

[7] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag Lecture Notes in Computer Science n°203, 1985.

[8] T. Coquand, G. Huet. "The Calculus of Constructions." Information and Computation, Volume 76, 1988.

[9] G. Dowek. "A Vernacular Syllabus." in [9]

[10] G. Dowek. "Naming and Scoping in a Mathematical Vernacular". To appear, Rapport de Recherche INRIA.

[11] G.Dowek. "A pattern matching algorithm for a restriction to the Calculus of Constructions." In preparation.

[12] J.Y. Girard. "Types and Proofs." translated and with appendices by P. Taylor and Y. Lafont. Cambrige University Press, 1989.

[13] L. Helmink. "Resolution and Type Theory." Proceedings of the ESOP Conference, Copenhagen, May 1990. Lecture Notes in Computer Science n°432. Springer-Verlag, 1990.

[14] G. Huet Ed. "The Calculus of Constructions, Documentation and user's guide." INRIA, Rapport Technique n°110, August 1989.

[15] G. Huet. “The Constructive Engine.” in “A Perspective in Theoretical Computer Science, Commemorative Volume for Gift Siromoney.” ed. R. Narasimhan, World Scientific Publishing, 89. also in [9].

[16] G. Huet. “A Uniform Approach to Type Theory.” Rapport de recherche INRIA n°795 , Février 88, also in “Logical Foundations of Functional Programming.” Addison-Wesley, 1990.

[17] G. Huet. “Résolution d'équations dans les langages d'ordres 1,2, ..., ω.” Thèse de Doctorat d'État, Université de Paris VII, 1976.

[18] G. Huet. “A Unification Algorithm for typed λ-calculus.” Theoretical Computer Science I, 1975, 27-57

[19] G.Huet. “Constrained Resolution. A Complete Method for Higher Order Logic.” Phd Thesis, Case Western Reserve University, 1972.

[20] G. Huet, B. Lang. “Proving and Applying Program Transformations Expressed with Second Order Patterns.” Acta Informatica 11, p. 31-55, 1978.

[21] D.C.Jensen, T. Pietrzykowski. “Mechanizing ω-order type theory through unification.” Theoretical Computer Science 3, 1976, 123-171

[22] D.A. Miller. “Unification Under Mixed Prefix.” Personal Communication.

[23] Ch. Paulin-Mohring. “Extraction de programmes dans le Calcul des Constructions.” Thèse, Université Paris VII, 1989.

[24] Ch. Paulin-Mohring. “Extracting Fω programs from proofs in the Calculus of Construction.” Proceedings of POPL 1989.

[25] D. Simon. “Checking Natural Language Proofs.” 9th International Conference on Automated Deduction. Argonne, Illinois, USA, May 1988, Lecture Notes in Computer Science n°310

GH from constable

Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics

Peter Dybjer

Chalmers University of Technology

(Draft)

Abstract

Martin-Löf's type theory is presented in several steps. The kernel is a dependently typed λ-calculus. Then there are schemata for inductive sets and families of sets and for primitive recursive functions and families of functions. Finally, there are set formers (generic polymorphism) and universes. At each step syntax, inference rules, and set-theoretic semantics are given.

1 Introduction

Usually Martin-Löf's type theory is presented as a closed system with rules for a finite collection of set formers. But it is also often pointed out that the system is in principle open to extension: we may introduce new sets when there is a need for them. The principle is that a set is by definition inductively generated - it is defined by its introduction rules, which are rules for generating its elements. The elimination rule is determined by the introduction rules and expresses definition by primitive recursion on the way the elements of the set are generated. (In this paper I shall use the term primitive recursive for the kind of recursion you have in type theory, which includes primitive recursive functionals and 'structural' recursion on an arbitrary inductive (=inductively defined) set (or family) including transfinite recursion.)

Backhouse [3] et.al. [4] exhibited a schema for *inductive sets* which delimits a class of definitions admissible in Martin-Löf's type theory including all the standard operations for forming small sets except the equality set. This schema extends Schroeder-Heister's schema for the logical constants [13, 14] to the type-theoretic case, where proof objects are explicitly represented in the theory.

Coquand and Paulin [5] and Dybjer [7] extended Backhouse's schema to incorporate *inductive families* and thus also *inductive predicates.* (Coquand and Paulin presented their schema as an extension of impredicative higher order logic and the calculus of constructions, but the formal pattern is much the same as the one in Dybjer [7].) This schema covers all the standard operations for forming small sets including the equality set. It also subsumes Martin-Löf's schema for inductive predicates in predicate logic [8].

In this paper I give a somewhat different presentation of the schema. One difference is that also definitions of functions by primitive recursion are presented schematically (much like in Martin-Löf [9]) rather than by the usual kind of elimination rules. I also separate the presentation of the

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks" and Styrelsen för Teknisk Utveckling.

process of inductive generation of sets and families from the process of introducing parameters (generic polymorphism). Moreover, I present a version of type theory without a logical framework and without an underlying theory of expressions (like in Martin-Löf's presentations of type theory before and including the book [10]).

I also show how to interpret type theory, with the schema, in classical set theory. This gives a non-intended but useful interpretation, compare Troelstra [16, page 2]: 'The simplest interpretation of $\mathbf{ML_0}$ is in terms of a hierarchy within classical set theory, where Π, Σ, etc, correspond to the formation of cartesian products, disjoint unions etc. as already indicated above; function, i.e., elements of cartesian products are regarded as equal if for each argument their values are equal, etc.'

Salvesen [12] presented details of such an interpretation of type theory. Well-orderings and the first universe were interpreted as inductively defined sets obtained by iterations of continuous operators.

Coquand and Paulin [5] proposed to give a set-theoretic interpretation of their schema for *inductive sets* by (i) translating the introduction rules defining a set to a strictly positive set operator in type theory; (ii) introducing rules for fixed points of such set operators in type theory and showing that the corresponding rules of the schema can be derived; (iii) interpreting type-theoretic strictly positive operators as ω_n-continuous functors on the category of sets (assuming a theory without universes).

In this paper I use Aczel's [1] notion of rule set rather than continuous functors. It is really only a variation, since a rule set generates a continuous operator. But it allows a direct concrete translation of the type-theoretic introduction rules to set-theoretic rule sets and generalizes the concrete construction of the term algebra T_Σ on a first order signature Σ.

I also interpret *inductive families.*

Even though the interpretation is a non-intended one, there is an analogy with Martin-Löf's intuitive justifications of the rules of type theory, whereby the formation rule receives its meaning from the introduction rules and the elimination rule receives its meaning from the equality rules.

I would also like to mention that Aczel [2] has shown how to interpret certain inductive sets, such as the well-orderings of type theory, in a *constructive set theory* (which itself can be interpreted in Martin-Löf's type theory). Does it follow that the whole of Martin-Löf's type theory can be interpreted in this constructive set theory?

Type theory is presented in the following steps.

- The dependently typed λ-calculus (section 2). This is like the simply typed λ-calculus with Π instead of $\rightarrow$. It consists essentially of the general rules and the rules for Π in Martin-Löf [10] except η-conversion.

- Schema for inductive sets (section 3). This part of the schema is closely related to the well-orderings.

- Schema for primitive recursive function definitions (section 4).

- Schema for inductive families (section 5). This generalizes the schema for inductive sets. The simpler case is presented separately for the purpose of the presentation only.

- Schema for primitive recursive families of functions (section 6). This generalizes the schema for primitive recursive functions.

- Generic set formers. The fact that a definition may depend on parameters gives rise to set formers or generic polymorphism (section 7). Typical ambiguity is also discussed briefly, and it is noted that the interpretation allows polymorphic constructors, but not polymorphic recursive functions. Moreover, the possibilty of internalizing the schema is discussed.

- Universes (section 8).

At each step I first give syntax, then inference rules, and finally a set-theoretic interpretation.

In this paper I don't discuss simultaneous induction and recursion. The reader is referred to Dybjer [7] for this and also for some examples of what can be defined using the schema.

2 The dependently typed λ-calculus

We use ordinary notation, but omit mentioning variable restrictions, etc.

2.1 Expressions

Set expressions:

$$A \quad ::= \quad \Pi x : A_0.A_1[x].$$

Element expressions:

$$a \quad ::= \quad x \mid \lambda x : A.a[x] \mid a_1(a_0).$$

Context expressions:

$$\Gamma \quad ::= \quad \epsilon \mid \Gamma, x : A.$$

Judgement expressions:

$$J \quad ::= \quad \Gamma\ context \mid \Gamma \vdash A\ set \mid \Gamma \vdash a : A \mid \Gamma \vdash A = A' \mid \Gamma \vdash a = a' : A.$$

2.2 Inference rules

Some premises are omitted.

General rules:

$$\epsilon\ context \qquad \frac{\Gamma\ context \qquad \Gamma \vdash A\ set}{\Gamma, x : A\ context}$$

$$\frac{\Gamma \vdash A\ set}{\Gamma \vdash A = A} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash a = a : A}$$

$$\frac{\Gamma \vdash A = A'}{\Gamma \vdash A' = A} \qquad \frac{\Gamma \vdash a = a' : A}{\Gamma \vdash a' = a : A}$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma \vdash A' = A''}{\Gamma \vdash A = A''} \qquad \frac{\Gamma \vdash a = a' : A \qquad \Gamma \vdash a = a'' : A}{\Gamma \vdash a = a'' : A}$$

$$\frac{\Gamma \vdash A = A' \qquad \Gamma \vdash a : A}{\Gamma \vdash a : A'} \qquad \frac{\Gamma \vdash A = A' \qquad \Gamma \vdash a = a' : A}{\Gamma \vdash a = a' : A'}$$

$$\frac{\Gamma \vdash A\ set}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash A_0\ set \qquad \Gamma \vdash A_1\ set}{\Gamma, x : A_0 \vdash A_1\ set} \qquad \frac{\Gamma \vdash A_0\ set \qquad \Gamma \vdash a : A_1}{\Gamma, x : A_0 \vdash a : A_1}$$

Rules for the cartesian product of a family of sets:

$$\frac{\Gamma \vdash A_0\ set \qquad \Gamma, x : A_0 \vdash A_1[x]\ set}{\Gamma \vdash \Pi x : A_0.A_1[x]\ set} \qquad \frac{\Gamma \vdash A_0 = A_0' \qquad \Gamma, x : A_0 \vdash A_1[x] = A_1'[x]}{\Gamma \vdash \Pi x : A_0.A_1[x] = \Pi x : A_0'.A_1'[x]}$$

$$\frac{\Gamma, x : A_0 \vdash a[x] : A_1}{\Gamma \vdash \lambda x : A_0.a[x] : \Pi x : A_0.A_1[x]} \qquad \frac{\Gamma, x : A_0 \vdash a[x] = a'[x] : A_1}{\Gamma \vdash \lambda x : A_0.a[x] = \lambda x : A_0.a'[x] : \Pi x : A_0.A_1[x]}$$

$$\frac{\Gamma \vdash a_1 : \Pi x : A_0.A_1[x] \qquad \Gamma \vdash a_0 : A_0}{\Gamma \vdash a_1(a_0) : A_1[a_0]} \qquad \frac{\Gamma \vdash a_1 = a_1' : \Pi x : A_0.A_1[x] \qquad \Gamma \vdash a_0 = a_0' : A_0}{\Gamma \vdash a_1(a_0) = a_1'(a_0') : A_1[a_0]}$$

$$\frac{\Gamma, x : A_0 \vdash a_1[x] : A_1[x] \qquad \Gamma \vdash a_0 : A_0}{\Gamma \vdash (\lambda x : A_0.a_1[x])(a_0) = a_1[a_0] : A_1[a_0]}$$

2.3 Interpretation of expressions

Let $[\![a]\!]\rho$ be the denotation of the expression a under the assignment ρ. This assigns a set to each variable in a finite list of variables which includes all variables which are free in a. Let $\emptyset$ be the empty assignment and let ρ_x^u abbreviate $\rho \cup \{\langle x, u\rangle\}$. Let also $[\![a]\!]$ abbreviate $[\![a]\!]\emptyset$.

The interpretation function is partial. Partiality is introduced in the interpretation of application. But the interpretation of a derivable judgement will be defined and true.

The method with a partial interpretation function has also been used by Streicher for a categorical interpretation of the calculus of constructions [15].

Interpretation of set expressions:

$$[\![\Pi x : A_0.A_1[x]]\!]\rho \quad = \quad \prod_{u \in [\![A_0]\!]\rho} [\![A_1[x]]\!]\rho_x^u.$$

This is defined iff $[\![A_0]\!]\rho$ is defined and $[\![A_1[x]]\!]\rho_x^u$ is defined whenever $u \in [\![A_0]\!]\rho$.

Interpretation of element expressions:

$$[\![x]\!]\rho \quad = \quad \rho(x).$$

This is always defined.

$$[\![\lambda x : A.a[x]]\!]\rho \quad = \quad \{\langle u, [\![a[x]]\!]\rho_x^u\rangle | u \in [\![A]\!]\rho\}.$$

This is defined iff $[\![A]\!]\rho$ is defined and $[\![a[x]]\!]\rho_x^u$ is defined whenever $u \in [\![A]\!]\rho$.

$$[\![a_1(a_0)]\!]\rho \quad = \quad ([\![a_1]\!]\rho)([\![a_0]\!]\rho).$$

This is defined iff $[\![a_1]\!]\rho$ and $[\![a_0]\!]\rho$ are defined, and $[\![a_1]\!]\rho$ is a function the domain of which contains $[\![a_0]\!]\rho$. (Observe that it is possible to interpret polymorphic application in set theory. This is not the case for interpretations of type theory in general, compare Streicher [15].)

Interpretation of context expressions:

$$[\![\epsilon]\!] \quad = \quad \{\emptyset\}.$$

This is always defined.

$$[\![\Gamma, x : A]\!] \quad = \quad \{\rho_x^u | \rho \in [\![\Gamma]\!] \wedge u \in [\![A]\!]\rho\}.$$

This is defined iff Γ is defined and $[\![A]\!]\rho$ is defined whenever $\rho \in [\![\Gamma]\!]$.

Interpretation of judgement expressions:

$$[\![\Gamma \ context]\!] \quad \text{iff} \quad [\![\Gamma]\!] \text{ is a set of assignments.}$$

This is defined iff $[\![\Gamma]\!]$ is defined.

$$[\![\Gamma \vdash A \ set]\!] \quad \text{iff} \quad [\![A]\!]\rho \text{ is a set whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![A]\!]\rho$ is defined whenever $\rho \in [\![\Gamma]\!]$.

$$[\![\Gamma \vdash a : A]\!] \quad \text{iff} \quad [\![a]\!]\rho \in [\![A]\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![a]\!]\rho$ and $[\![A]\!]\rho$ are defined whenever $\rho \in [\![\Gamma]\!]$.

$$[\![\Gamma \vdash A = A']\!] \quad \text{iff} \quad [\![A]\!]\rho = [\![A']\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![A]\!]\rho$ and $[\![A']\!]\rho$ are defined whenever $\rho \in [\![\Gamma]\!]$.

$$[\![\Gamma \vdash a = a' : A]\!] \quad \text{iff} \quad [\![a]\!]\rho = [\![a']\!]\rho \wedge [\![a]\!]\rho \in [\![A]\!]\rho \wedge [\![a']\!]\rho \in [\![A]\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

This is defined iff $[\![\Gamma]\!]$ is defined and if $[\![a]\!]\rho$, $[\![a']\!]\rho$, and $[\![A]\!]\rho$ are defined whenever $\rho \in [\![\Gamma]\!]$.

2.4 Soundness of the inference rules

Checking the soundness of the inference rules means checking that the interpretation of the conclusion of a rule is defined and true whenever the interpretation of the premises are defined and true.

It is quite straightforward to check the soundness of all the inference rules. As an illustration we show the soundness of the rule of application. The premises are interpreted as

$$[\![a_1]\!]\rho \in \prod_{u \in [\![A_0]\!]\rho} [\![A_1[x]]\!]\rho_x^u \text{ whenever } \rho \in [\![\Gamma]\!]$$

and

$$[\![a_0]\!]\rho \in [\![A_0]\!]\rho \text{ whenever } \rho \in [\![\Gamma]\!].$$

From this we conclude that

$$([\![a_1]\!]\rho)([\![a_0]\!]\rho) \in [\![A_1[x]]\!]\rho_x^{[\![a_0]\!]\rho} \text{ whenever } \rho \in [\![\Gamma]\!],$$

and hence the conclusion of the rule follows, since

$$[\![A_1[x]]\!]\rho_x^{[\![a_0]\!]\rho} = [\![A_1[a_0]]\!]\rho.$$

2.5 Telescopes

In the description of the schema below we shall frequently refer to sequences of dependent sets, to sequences (tuples) of elements, and to sequences of typings of elements. De Bruijn has introduced the term *telescope* for such sequences of dependent sets. Telescopes are closely related to contexts, they are so as to speak contexts treated as objects. Telescopes can also be viewed as obtained by iterating the Σ-construction.

It is intended that the reader view the terms telescope, tuple, etc., and certain associated notation as abbreviations and reduce them to formal notions of type theory in a way to be suggested below. (The description is not complete, and there are places where the notation needs to be interpreted with a certain amount of good will in order to make good sense.)

The new notation is explained as follows:

- As is a telescope means that A_1 set, $A_2[x_1]$ set $(x_1 : A_1), \ldots, A_n[x_1, \ldots, x_{n-1}]$ $(x_1 : A_1, \ldots, x_{n-1} : A_{n-1})$;
- $As = As'$ means that $A_1 = A_1$ set, $A_2[x_1] = A_2[x_1]$ set $(x_1 : A_1), \ldots, A_n[x_1, \ldots, x_{n-1}] = A'_n[x_1, \ldots, x_{n-1}]$ $(x_1 : A_1, \ldots, x_{n-1} : A_{n-1})$;
- $as :: As$ means that $a_1 : A_1, a_2 : A_2[a_1], \ldots, a_n : A_n[a_1, \ldots, a_{n-1}]$;
- $as = as' :: As$ means that $a_1 = a'_1 : A_1, a_2 = a'_2 : A_2[a_1], \ldots, a_n = a'_n : A_n[a_1, \ldots, a_{n-1}]$

respectively.

We also write $f(as)$ for $f(a_1, \ldots, a_n)$, $f(as, bs)$ for $f(a_1, \ldots, a_n, b_1, \ldots, b_m)$, etc.

An alternative approach would be to extend type theory with *formal* notions of telescopes and tuples. In addition to the standard forms of judgement we would have the new forms As is a telescope; $As = As'$; $as :: As$; $as = as' :: As$. As other forms of judgement these judgements would be made under assumptions. We would then have suitable rules for forming telescopes and tuples. Furthermore, if we have telescopes, a context can be viewed as a single assumption $xs :: As$. Compare also the discussion in section 7 on internalization of the schema.

The set-theoretic semantics can be extended to telescopes and tuples. When we write $us \in [\![As]\!]$, we understand that us is a tuple $\langle u_1, \ldots, u_n \rangle$ such that $u_1 \in [\![A_1]\!], \ldots, u_n \in [\![A_n[x_1, \ldots, x_{n-1}]]\!]^{u_1 \ldots u_{n-1}}_{x_1 \ldots x_{n-1}}$.

We also use index notation such as $(a_k)_k$, $(A_k)_k$, and $(a_k : A_k)_k$ to stand for $a_1, \ldots, a_n$, $A_1, \ldots, A_n$, and $a_1 : A_1, \ldots, a_n : A_n$ respectively. This makes it possible, for example, to talk about *non-dependent* telescopes of the form $(A_k)_k$.

3 Schema for inductive sets

We have now presented the syntax and rules of the dependently typed λ-calculus. Call this theory T_0. T_0 can be extended successively obtaining the theories $T_1, T_2, \ldots$.

There are two different kinds of extensions.

The first kind is when T_{n+1} is obtained from $T = T_n$ by adding a new set former P. Such a set former is defined inductively and specified by its formation and introduction rules (see section 3, 5, and 7).

The second kind is when T_{n+1} is obtained from $T = T_n$ by adding a new function constant f, which is defined by primitive recursion on some set (or family) and is specified by its type and its computation rules (see section 4, 6, and 7). In this way we get schematic elimination and equality rules.

We first treat the simple case without parameters and inductive families.

3.1 Expressions

Set expressions:

$$A ::= P.$$

Element expressions

$$a ::= intro_i(as, (b_k)_k).$$

3.2 Inference rules

(J abbreviates $\Gamma \vdash J$.)
Formation rules:

$$P \ set,$$

$$P = P.$$

The ith *introduction rules*:

$$\frac{as :: Gs_i \qquad (b_k : Hs_{ik}[as] \to P)_k}{intro_i(as, (b_k)_k) : P},$$

$$\frac{as = as' :: Gs_i \qquad (b_k = b'_k : Hs_{ik}[as] \to P)_k}{intro_i(as, (b_k)_k) = intro_i(as', (b'_k)_k) : P},$$

where

- Gs_i is a telescope relative to T;
- $Hs_{ik}[xs]$ is a telescope relative to T in the context $xs :: Gs_i$ for each k.

3.3 Inductive sets in set theory

We shall use Aczel's [1] set-theoretic notion of rule set to interpret the introduction rules for a new set former. The set defined inductively by a rule set is the least set closed under all rules in the rule set.

A *rule* on a base set U in Aczel's sense is a pair of sets $\langle u, v\rangle$, often written

$$\frac{u}{v},$$

such that $u \subseteq U$ and $v \in U$.

Let Φ be a set of rules on U.

A set w is Φ-*closed* if

$$\frac{u}{v} \in \Phi \wedge u \subseteq w \supset v \in w.$$

There is a least Φ-closed set

$$\mathcal{I}(\Phi) = \bigcap\{w \subseteq U | w \ \Phi - closed\},$$

the set inductively defined by Φ.

3.4 Interpretation of expressions

Interpretation of set expressions:

$$[\![P]\!]\rho \quad = \quad \mathcal{I}(\Phi_P),$$

where

$$\Phi_P = \bigcup_i \{ \frac{\bigcup_k \operatorname{ran} v_k}{\langle |\mathit{intro}_i|, us, (v_k)_k \rangle} | us \in [\![Gs_i]\!], (v_k \in [\![Hs_{ik}[xs]]\!]^{us}_{xs} \to U)_k \},$$

where $|\mathit{intro}_i| \in \omega$ is a code for the constructor intro_i, and $U = V_\alpha$, the set of sets generated before stage α in the cumulative hierarchy, where α is chosen so that Φ_P is a rule set on U. This induces the following requirements on the ordinal α:

- V_α is closed under tupling, that is, α is a limit ordinal;
- $\omega \subseteq V_\alpha$, that is, $\omega \leq \alpha$;
- $[\![Gs_i]\!] \subseteq V_\alpha$ for all i;
- $[\![Hs_{ik}[xs]]\!]^{us}_{xs} \to V_\alpha \subseteq V_\alpha$ for all $us \in [\![Gs_i]\!]$ and all ik. This can be achieved if $[\![Hs_{ik}[xs]]\!]^{us}_{xs} \subseteq V_\alpha$ and if $\operatorname{card} [\![Hs_{ik}[xs]]\!]^{us}_{xs} < \operatorname{card} \alpha$ for all $us \in [\![Gs_i]\!]$ and all ik. Because assume that $v_k \in [\![Hs_{ik}[xs]]\!]^{us}_{xs} \to V_\alpha$. Then for each $ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}$ we have $\langle ws, v_k(ws) \rangle \in V_{\beta_{ws}}$ for some ordinal $\beta_{ws} < \alpha$. Then $\beta = \sup_{ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}} \beta_{ws} < \alpha$. Hence, $v_k = \{\langle ws, v_k(ws) \rangle | ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}\} \subseteq V_\beta$. So $v_k \in V_\alpha$.

(If the theory T does not include universes, then we can find $\alpha < \omega_\omega$ (assuming $\omega_{\alpha+1} = 2^{\omega_\alpha}$), because the rank and cardinality of the interpretation of sets constructed without universes are $< \omega_\omega$.)

Interpretation of element expressions:

$$[\![\mathit{intro}_i(as, (b_k)_k)]\!]\rho \quad = \quad \langle |\mathit{intro}_i|, [\![as]\!]\rho, ([\![b_k]\!]\rho)_k \rangle.$$

3.5 Soundness of the inference rules

Formation rule. We have already shown that $[\![P]\!]\rho$ is a set.

The introduction rule is sound because assume that $us \in [\![Gs_i]\!]\rho$ whenever $\rho \in [\![\Gamma]\!]$, and $v_k \in [\![Hs_{ik}[xs]]\!]\rho^{us}_{xs} \to [\![P]\!]$, whenever $\rho \in [\![\Gamma]\!]$, for all k. Then it follows that $\bigcup_k \operatorname{ran} v_k \subseteq [\![P]\!]$ and hence since $[\![P]\!]$ is Φ_P-closed $\langle |\mathit{intro}_i|, us, (v_k)_k \rangle \in [\![P]\!]$.

3.6 Logical consistency

Absurdity ($\perp$) is interpreted as the empty set, the set which is defined by the empty list of introduction rules. We get that $[\![\perp]\!] = \emptyset$, so we cannot have $a : \perp$, since this entails $[\![a]\!] \in \emptyset$. Hence the set-theoretic interpretation shows the logical consistency of Martin-Löf's type theory.

4 Primitive recursive functions

Functions can be defined by recursion on the way the elements of P are generated (primitive or structural recursion). Here we give a schema for such defintions rather than a single elimination rule.

4.1 Syntax

$$a ::= f.$$

4.2 Inference rules

Elimination rules:

$$f : \Pi z : P.C[z],$$
$$f = f : \Pi z : P.C[z].$$

The ith *equality rules*:

$$\frac{as :: Gs_i \qquad (b_k : Hs_{ik}[as] \to P)_k}{f(intro_i(as, (b_k)_k)) = d_i(as, (b_k, \lambda zs :: Hs_{ik}[as].f(b_k(zs)))_k) : C[intro_i(as, (b_k)_k)]},$$

where

- $C[z]$ is a set in the context $z : P$;
- $$\begin{aligned} d_i \quad : \quad & \Pi xs :: Gs_i. \\ & (\Pi y_k : (Hs_{ik}[xs] \to P). \\ & \ \Pi y'_k : (\Pi zs :: Hs_{ik}[xs].C[y_k(zs)]))_k. \\ & C[intro_i(xs, (y_k)_k)]. \end{aligned}$$

4.3 Primitive recursive functions in set theory

A rule set Φ is *deterministic* if

$$\frac{u}{v} \in \Phi \wedge \frac{u'}{v} \in \Phi \supset u = u'.$$

If Φ is deterministic functions on $\mathcal{I}(\Phi)$ can be defined by recursion on the way the elements in $\mathcal{I}(\Phi)$ are generated.

4.4 Interpretation of expressions

Since Φ_P is deterministic, type-theoretic primitive recursion can be interpreted as set-theoretic primitive recursion on $\mathcal{I}(\Phi_P)$. Let

$$[\![f]\!]\rho \quad = \quad \mathcal{I}(\Psi_f),$$

where

$$\begin{aligned} \Psi_f \quad = \quad & \bigcup_i \{ \frac{\bigcup_k \{ \langle v_k(ws), v'_k(ws) \rangle | ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs} \}}{\langle \langle |intro_i|, us, (v_k)_k \rangle, [\![d_i]\!](us, (v_k, v'_k)_k) \rangle} | \\ & us \in [\![Gs_i]\!], \\ & (v_k \in [\![Hs_{ik}[xs]]\!]^{us}_{xs} \to [\![P]\!], \\ & v'_k \in \prod_{ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}} [\![C[y_k(zs)]]\!]^{ws\, v_k}_{zs\, y_k})_k \}. \end{aligned}$$

Ψ_f is a rule set on $\sum_{w\in[\![P]\!]}[\![C[z]]\!]^w_z$. This is easily proved by Φ_P-induction since the pairs in $\mathcal{I}(\Psi_f)$ are generated in parallel with the elements of $\mathcal{I}(\Phi_P)$ and since the requirements on d_i ensure that the second component is in $[\![C[z]]\!]^w_z$.

4.5 Soundness of the inference rules

Since Φ_P is deterministic Ψ_f defines a function on $[\![P]\!]$. Hence

$$[\![f]\!]\rho \in \prod_{w\in[\![P]\!]\rho} [\![C[z]]\!]\rho^w_z,$$

and the elimination rule is validated.

To prove the soundness of the equality rules we need to prove three things: two memberships and an equality. The memberships are immediate. The equality is a direct consequence of the definition of Ψ_f.

5 Inductive families

We now treat the more general case of inductively defined families of sets, but still postpone the discussion of parameters.

5.1 Expressions

Set expressions:

$$A ::= P(as).$$

Element expressions

$$a ::= \mathit{intro}_i(as, (b_k)_k).$$

5.2 Inference rules

Formation rules:

$$\frac{as :: Is}{P(as)\ set},$$

$$\frac{as :: Is}{P(as) = P(as)},$$

where

- Is is a telescope relative to T.

The ith *introduction rules*:

$$\frac{as :: Gs_i \qquad (b_k : \Pi zs :: Hs_{ik}[as].P(qs_{ik}[as, zs]))_k}{\mathit{intro}_i(as, (b_k)_k) : P(ps_i[as])},$$

$$\frac{as = as' :: Gs_i \qquad (b_k = b'_k : \Pi zs :: Hs_{ik}[as].P(qs_{ik}[as, zs]))_k}{\mathit{intro}_i(as, (b_k)_k) = \mathit{intro}_i(as', (b'_k)_k) : P(ps_i[as])},$$

where

- Gs_i is a telescope relative to T;
- $Hs_{ik}[xs]$ is a telescope relative to T in the context $xs :: Gs_i$ for each k;
- $qs_{ik}[xs, zs] :: Is$ relative to T in the context $xs :: Gs_i, zs :: Hs_{ik}[xs]$ for each k;
- $ps_i[xs] :: Is$ relative to T in the context $xs :: Gs_i$.

5.3 Inductive families in set theory

Let I and U be sets and let Φ be a rule set on $I \times U$. Then Φ inductively defines a family $\mathcal{IF}(\Phi)$ of sets in U over I by

$$\mathcal{IF}(\Phi)(i) = \{u \in U | \langle i, u\rangle \in \mathcal{I}(\Phi)\}$$

for each $i \in I$.

5.4 Interpretation of expressions

Interpretation of set expressions:

$$[\![P(as)]\!]\rho \quad = \quad \mathcal{IF}(\Phi_P)([\![as]\!]\rho),$$

where

$$\begin{aligned}\Phi_P \quad = \quad \bigcup_i \{ & \frac{\bigcup_k \{\langle [\![qs_{ik}[xs, zs]]\!]^{us\,ws}_{xs\,zs}, v_k(ws)\rangle | ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}\}}{\langle [\![ps_i[xs]]\!]^{us}_{xs}, \langle |intro_i|, us, (v_k)_k\rangle\rangle} | \\ & us \in [\![Gs_i]\!], \\ & (v_k \in [\![Hs_{ik}[xs]]\!]^{us}_{xs} \to U)_k\},\end{aligned}$$

where U is chosen so that Φ_P is a rule set on $[\![Is]\!] \times U$. Such a $U = V_\alpha$ is found in a similar way to the case for inductive sets above.

Interpretation of element expressions:

$$[\![intro_i(as, (b_k)_k)]\!]\rho \quad = \quad \langle |intro_i|, [\![as]\!]\rho, ([\![b_k]\!]\rho)_k\rangle.$$

5.5 Soundness of the inference rules

Formation rule. This is sound since $\mathcal{IF}(\Phi_P)$ is a family of sets over $[\![Is]\!]$.

The introduction rule is sound because assume that

$$us \in [\![Gs_i]\!]\rho$$

and

$$v_k \in \prod_{ws \in [\![Hs_{ik}[xs]]\!]\rho^{us}_{xs}} [\![P(qs_{ik}[xs, zs])]\!]\rho^{us\,ws}_{xs\,zs},$$

whenever $\rho \in [\![\Gamma]\!]$, for all k. Then it follows that for each $ws \in [\![Hs_{ik}[xs]]\!]\rho^{us}_{xs}$ we have

$$\langle [\![qs_{ik}[xs, zs]]\!]\rho^{us\,ws}_{xs\,zs}, v_k(ws)\rangle \in \mathcal{I}(\Phi_P).$$

Hence, by Φ_P-closedness

$$\langle [\![ps_i[xs]]\!]\rho^{us}_{xs}, \langle |intro_i|, us, (v_k)_k\rangle\rangle \in \mathcal{I}(\Phi_P)$$

and thus

$$\langle |intro_i|, us, (v_k)_k\rangle \in [\![P(ps_i[xs])]\!].$$

6 Primitive recursive families of functions

We give a schema for functions which are defined by recursion on the way the elements of $P(as)$ are generated. This generalizes the schema in section 4. Note that we have a kind of simultaneous recursion: an element of $P(ps_i[as])$ is generated from the elements of $(P(qs_{ik}[as, zs]))_k$.

6.1 Syntax

$$a ::= f(as)$$

6.2 Inference rules

Elimination rule:

$$\frac{as :: Is}{f(as) : \Pi z : P(as).C[z]},$$

$$\frac{as = as' :: Is}{f(as) = f(as') : \Pi z : P(as).C[z]}.$$

The ith equality rule:

$$\frac{as :: Gs_i \qquad (b_k : \Pi zs :: Hs_{ik}[as].P(qs_{ik}[as, zs]))_k}{\begin{array}{ll} & f(ps_i[as])(intro_i(as, (b_k)_k)) \\ = & d_i(as, (b_k, (\lambda zs :: Hs_{ik}[as].f(qs_{ik}[as, zs])(b_k(zs))))_k) \\ : & C[ps_i[as], intro_i(as, (b_k)_k)] \end{array}},$$

where

- $C[xs, z]$ is a set in the context $xs :: Is, z : P(xs)$;
-
$$\begin{array}{lll} d_i & : & \Pi xs :: Gs_i. \\ & & (\Pi y_k : (\Pi zs :: Hs_{ik}[xs].P(qs_{ik}[xs, zs])). \\ & & \ \Pi y'_k : (\Pi zs :: Hs_{ik}[xs].C[qs_{ik}[xs, zs], y_k(zs)]))_k. \\ & & C[ps_i[xs], intro_i(xs, (y_k)_k)]. \end{array}$$

6.3 Primitive recursive families of functions in set theory

The rule set Φ_P is still deterministic. As a consequence we could define functions on the pairs $\langle as, c\rangle$ in $\mathcal{I}(\Phi_P)$. But we want curried versions instead. Such can be defined as inductive families of set-theoretic functions.

6.4 Interpretation of expressions

Let

$$[\![f(as)]\!]\rho \quad = \quad \mathcal{IF}(\Psi_f)([\![as]\!]\rho),$$

where

$$\begin{aligned}\Psi_f \quad = \quad & \bigcup_i \{ \frac{\bigcup_k \{\langle [\![qs_{ik}[xs,zs]]\!]^{us\,ws}_{xs\,zs}, \langle v_k(ws), v'_k(ws)\rangle\rangle | ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}\}}{\langle [\![ps_i[xs]]\!]^{us}_{xs}, \langle\langle |intro_i|, us, (v_k)_k\rangle, [\![d_i]\!](us, (v_k, v'_k)_k)\rangle\rangle} | \\ & us \in [\![Gs_i]\!], \\ & (v_k \in \prod_{ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}} [\![P(qs_{ik}[xs,zs])]\!]^{us\,ws}_{xs\,zs}, \\ & \;\; v'_k \in \prod_{ws \in [\![Hs_{ik}[xs]]\!]^{us}_{xs}} [\![C[qs_{ik}[xs,zs], y_k(zs)]]\!]^{us\,ws\,v_k}_{xs\,zs\,y_k})_k\}.\end{aligned}$$

Ψ_f is a rule set on $\sum_{us \in [\![Is]\!]} \sum_{w \in [\![P(xs)]\!]^{us}_{xs}} [\![C[xs,z]]\!]^{us\,w}_{xs\,z}$.

6.5 Soundness of inference rules

Omitted.

7 Polymorphism

7.1 Generic set formers and parameters

So far we have presented a completely monomorphic version of type theory similar to the type theory presented in Martin-Löf [9]. (Martin-Löf went one step further and also introduced a new constant P for each instance of $\Pi x : A_0.A_1[x]$ and a new constant f for each instance $\lambda x : A.a[x]$. Thereby bound variables were avoided altogether.) But it is important both for convenience and expressiveness to introduce parameters in the inductive definitions. (We distinguish *generic set formers*, the arguments of which are called parameters, and *inductive families of sets*, the arguments of which are called indices.) A parameter can be either a set (or family) or an element (or function). The standard set formers all have sets (or families) as parameters: these are the A, A_0, and A_1 in $\Sigma x : A_0.A_1$, $A_0 + A_1$, $Eq(A)$, $Wx : A_0.A_1$. A nice example of a set former which has an element as a parameter is the equality predicate (due to Christine Paulin) $Eq'(A,a)$ on a set A, where the set A and the element $a : A$ are parameters. The elimination rule (disregarding proof objects) for this predicate is the rule of substitution: if C is a predicate on A such that $C(a)$ is true, then $\forall z : A.(Eq'(A,a)(z) \supset C(z))$. This rule is a derived rule for $Eq(A)$, which has a more complex elimination rule expressing that $Eq(A)$ is defined as the smallest reflexive relation on A. Another example of a set former which takes an element as a parameter is Peterson's and Synek's trees (generalized well-orderings) [11].

The set-theoretic interpretation extends directly to the case with parameters.

7.2 Typical ambiguity

The set-theoretic interpretation given above is polymorphic (introduces typical ambiguity) in the constructors but not in the recursive functions. This is because the denotation of $intro_i(as,(b_k)_k)$

does not depend on the denotations of Gs_i, and $Hs_{ik}[xs]$, and (in the case of families) Is, whereas the denotation of f depends on the denotations of Gs_i, $Hs_{ik}[xs]$, and d_i, and (in the case of families) Is, $ps_i[xs]$, and $qs_{ik}[xs, zs]$.

7.3 Internalization

I have presented an open theory, that is, a theory which can be extended whenever there is a need for it. But the schema precisely determines what an admissible extension of a given theory T is, and hence the schema defines a collection of theories obtainable by such extensions.

Is it possible to turn this external schema into an internal construction? Consider first the case of *inductive sets*. Each set P is determined by a list (indexed by i) of pairs of telescopes Gs_i and lists (indexed by k) of telescopes Hs_{ik}. It is tempting to write something like

$$P = \mathcal{W}((xs :: Gs_i, (Hs_{ik}[xs])_k)_i)$$

and

$$f = \mathcal{T}((xs :: Gs_i, (Hs_{ik}[xs])_k, d_i)_i),$$

because of the similarity between the schema for inductive sets and structural recursive functions on the one hand and the rules for the well-orderings on the other. We could then put

$$Wx : A_0.A_1[x] = \mathcal{W}(((x : A_0), ((A_1[x])))),$$

and observe that the well-orderings is the special case where all lists and telscopes have length 1. Other standard set formers could be defined by

$$\begin{array}{rcl}
\bot & = & \mathcal{W}(()), \\
\top & = & \mathcal{W}(((), ())), \\
N & = & \mathcal{W}(((), ()); ((), (()))), \\
\mathcal{O} & = & \mathcal{W}(((), ()); ((), (())); ((), ((N)))), \\
A_0 + A_1 & = & \mathcal{W}(((A_0), ()); ((A_1), ())), \\
A_0 \times A_1 & = & \mathcal{W}(((A_0; A_1), ())), \\
\Sigma x : A_0.A_1[x] & = & \mathcal{W}(((x : A_0; A_1[x]), ())).
\end{array}$$

However, there are no formal means in type theory for introducing $\mathcal{W}$ and $\mathcal{T}$, not even in the theory of logical types (Martin-Löf's logical framework). It seems that we would need to extend this framework with certain formal notions of telescope and list.

Also note that we need the extra generality provided by the schema as compared with the well-orderings, since we consider *intensional* type theory. In *extensional* type theory on the other hand, we can use well-orderings for representing inductive sets, see Dybjer [6]. But even so, this is done by non-trivial coding and by assuming some basic set formers such as $\bot$, $\top$, $+$, and $\times$, in addition to $\rightarrow$, which is needed for the schema too.

For *inductive families* we could similarly try to write

$$P = \mathcal{WF}(Is, (xs :: Gs_i, (zs :: Hs_{ik}[xs], qs_{ik}[xs, zs])_k, ps_i[xs])_i)$$

and

$$f = \mathcal{TF}(Is, (xs :: Gs_i, (zs :: Hs_{ik}[xs], qs_{ik}[xs, zs])_k, ps_i[xs], d_i)_i).$$

This does not resemble any standard set former, even though the first three arguments have a similar function to the first three arguments of Petersson's and Synek's trees [11].

Some set formers written in terms of $\mathcal{WF}$:

$$\begin{array}{rcl} Eq(A) & = & \mathcal{WF}(A,((x:A),(),(x;x))), \\ Eq'(A,a) & = & \mathcal{WF}(A,((),(),(a))). \end{array}$$

8 Universes

We assume a countable sequence $U_0, U_1, U_2, \ldots$ and formulate rules for universes a la Russell [10].

Formation rules:

$$U_n \ set,$$

$$U_n = U_n.$$

Introduction rules:

$$U_m : U_n,$$

$$U_m = U_m : U_n$$

if $m < n$, and

$$\frac{\Gamma \vdash A_0 : U_n \qquad \Gamma, x : A_0 \vdash A_1[x] : U_n}{\Gamma \vdash \Pi x : A_0.A_1[x] : U_n},$$

$$\frac{\Gamma \vdash A_0 = A_0' : U_n \qquad \Gamma, x : A_0 \vdash A_1[x] = A_1'[x] : U_n}{\Gamma \vdash \Pi x : A_0.A_1[x] = \Pi x : A_0'.A_1'[x] : U_n}.$$

Elimination rules:

$$\frac{A : U_n}{A \ set},$$

$$\frac{A = A' : U_n}{A = A'}.$$

These rules extend the dependently typed λ-calculus with universes (getting the theory T_0^U) and are independent of the particular set formers introduced later. As in the case without universes we may extend T_0^U obtain a sequence of theories $T_1^U, T_2^U, \ldots$. When extending $T = T_n^U$ to T_{n+1}^U by adding a new set former P or a new function constant f we may use the rules for universes to justify that the Gs_i, $Hs_{ik}[xs]$, etc. are sets.

Moreover, for each set former $P(ts)$ which may depends on certain parameters ts (see the previous section) we have universe introduction rules reflecting the formation rule of P. Assume that the definition of $P(ts)$ involves the telescopes $Gs_i[ts]$ and $Hs_{ik}[xs, ts]$ (see section 3 and 5), and that $Gs_i[ts] : U_n$ and $Hs_{ik}[xs, ts] : U_n$ $(xs :: Gs_i)$ whenever the set parameters in ts are in U_n. Then we get an introduction rule for U_n by modifying the conclusion $P(ts)$ set of the formation rule to $P(ts) : U_n$, by modifying each premise for a set parameter $A : set$ to $A : U_n$, and by leaving each premise for an element unchanged. (This situation is somewhat complex and we won't give a completely formal presentation. If we introduce the internal set former for inductive families $\mathcal{WF}$, then it would be easy to let the formation rule of $\mathcal{WF}$ be reflected as a universe introduction rule.)

A natural set-theoretic interpretation of the sequence of universes is as a countable sequence of set-theoretic universes. But this would not reflect the fact that a universe in type theory is a set, and thus inductively defined by its introduction rules. So instead we could make the interpretation dependent on the particular collection of set formers introduced, and let rule sets corresponding to the introduction rules for U_n inductively generate $[\![U_n]\!]$. The latter approach is similar to Salvesen's interpretation of the universe [12]. Note however that, provided we have introduced at least one infinite set in U_0, the requirement that $[\![U_0]\!]$ is a set implies that there exists a strongly inaccessible cardinal $\sup_{u \in [\![U_0]\!]} \operatorname{card} u$.

References

[1] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter C.7, pages 739–782. North Holland, 1977.

[2] P. Aczel. The type theoretic interpretation of constructive set theory: inductive definitions. In *Logic, Methodology and Philosophy of Science VII*, pages 17–49. Elsevier Science Publishers B.V., 1986.

[3] R. Backhouse. On the meaning and construction of the rules in Martin-Löf's theory of types. Technical Report CS 8606, University of Groningen, Department of Mathematics and Computing Science, 1986. Presented at the Workshop on General Logic, Edinburgh, February 1987.

[4] R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory (part 1). *Formal Aspects of Computing*, 1:19–84, 1989.

[5] T. Coquand and C. Paulin. Inductively defined types. In *Proceedings of the Workshop on Programming Logic, Båstad*, May 1989.

[6] P. Dybjer. Inductively defined sets in Martin-Löf's type theory. In *Proceedings of the Workshop on General Logic, Edinburgh*, February 1987.

[7] P. Dybjer. An inversion principle for Martin-Löf's type theory. In *Proceedings of the Workshop on Programming Logic, Båstad*, May 1989.

[8] P. Martin-Löf. Hauptsatz for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland, 1971.

[9] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[10] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[11] K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science*, pages 128–140. Springer-Verlag, LNCS 389, 1989.

[12] A. B. Salvesen. Typeteori - en studie. Technical report, Department of Computer Science, University of Oslo, 1984. Cand.Scient-thesis.

[13] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4), December 1984.

[14] P. Schroeder-Heister. Judgements of higher levels and standardized rules for logical constants in Martin-Löf's theory of logic. Unpublished paper, June 1985.

[15] T. Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1988.

[16] A. S. Troelstra. On the syntax of Martin-Löf's type theories. *Theoretical Computer Science*, 51:1–26, 1987.

Encoding a Dependent-Type λ-Calculus in a Logic Programming Language

Amy Felty
INRIA Sophia-Antipolis
2004, Route des Lucioles
06565 Valbonne Cedex, France

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA

Abstract

Various forms of typed λ-calculi have been proposed as specification languages for representing wide varieties of object logics. The *logical framework*, LF, is an example of such a dependent-type λ-calculus. A small subset of intuitionistic logic with quantification over simply typed λ-calculus has also been proposed as a framework for specifying general logics. The logic of *hereditary Harrop* formulas with quantification at all non-predicate types, denoted here as hh^ω, is such a meta-logic that has been implemented in both the Isabelle theorem prover and the λProlog logic programming language. Both frameworks provide for specifications of logics in which details involved with free and bound variable occurrences, substitutions, eigenvariables, and the scope of assumptions within object logics are handled correctly and elegantly at the "meta" level. In this paper, we show how LF can be encoded into hh^ω in a direct and natural way by mapping the typing judgments in LF into propositions in the logic of hh^ω. This translation establishes a very strong connection between these two languages: the order of quantification in an LF signature is exactly the order of a set of hh^ω clauses, and the proofs in one system correspond directly to proofs in the other system. Relating these two languages makes it possible to provide implementations of proof checkers and theorem provers for logics specified in LF by using standard logic programming techniques which can be used to implement hh^ω.

1 Introduction

The design and construction of computer systems that can be used to specify and implement large collections of logics has been the goal of several different research projects. In this paper we shall focus on two approaches to designing such systems. One approach is based on the use of dependent-type λ-calculi as a meta-language while another approach is based on the use of a very simple intuitionistic logic as a meta-language. The Logical Framework (LF) [HHP87] and the Calculus of Constructions (CC) [5] are two examples of dependent-type calculi that have been proposed as meta-logics. The Isabelle theorem prover [Pau89] and the λProlog logic programming language [NM88] provide implementations of a common subset of intuitionistic logic, called hh^ω here, that can be used to specify a wide range of logics. Both Isabelle and λProlog can turn specifications of logics into proof checkers and theorem provers by making use of the unification of simply typed λ-terms and goal-directed, tactic-style search.

In this paper, we shall show that these two meta-languages are essentially of the same expressive power. This is done by showing how to translate LF specifications and judgments into a collection of hh^ω formulas such that correct typing in LF corresponds to intuitionistic provability in hh^ω. Besides

[1]To appear in the proceedings of the "10th International Conference on Automated Deduction," July 1990.

answering the theoretical question about the precise relationship between these meta-languages, this translation also describes how LF specifications of an object logic can be implemented using unification and goal-directed search since these techniques provide implementations of hh^ω.

In Section 2 we present the meta-logic hh^ω and in Section 3 we present LF. Section 4 presents a translation of LF into hh^ω and Section 5 contains a proof of its correctness. Section 6 provides examples of this translation and Section 7 concludes.

2 The Meta-Logic

Let S be a fixed, finite set of *primitive types* (also called *sorts*). We assume that the symbol o is always a member of S. Following Church [Chu40], o is the type for propositions. The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, denoted by the binary, infix symbol $\rightarrow$. The Greek letters τ and σ are used as syntactic variables ranging over types. The type constructor $\rightarrow$ associates to the right. If τ_0 is a primitive type then the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ has $\tau_1, \ldots, \tau_n$ as *argument types* and τ_0 as *target type*. The *order* of a primitive type is 0 while the order or a non-primitive type is one greater than the maximum order of its argument types.

For each type τ, we assume that there are denumerably many constants and variables of that type. Constants and variables do not overlap and if two constants (or variables) have different types, they are different constants (or variables). A *signature* is a finite set Σ of constants and variables whose types are such that their argument types do not contain o. A constant with target type o is a *predicate constant*. We often enumerate signatures by listing their members as pairs, written $a{:}\tau$, where a is a constant of type τ. Although attaching a type in this way is redundant, it makes reading signatures easier.

Simply typed λ-terms are built in the usual way. The logical constants are given the following types: $\wedge$ (conjunction) and $\supset$ (implication) are both of type $o \rightarrow o \rightarrow o$; $\top$ (true) is of type o; and $\forall_\tau$ (universal quantification) is of type $(\tau \rightarrow o) \rightarrow o$, for all types τ not containing o. A formula is a term of type o. The logical constants $\wedge$ and $\supset$ are written in the familiar infix form. The expression $\forall_\tau(\lambda z\ t)$ is written simply as $\forall_\tau z\ t$.

If x and t are terms of the same type then $[t/x]$ denotes the operation of substituting t for all free occurrences of x, systematically changing bound variables in order to avoid variable capture. The expression $[t_1/x_1, \ldots, t_n/x_n]$ will denote the simultaneous substitution of the terms $t_1, \ldots, t_n$ for the variables $x_1, \ldots, x_n$, respectively.

We shall assume that the reader is familiar with the usual notions and properties of α, β, and η conversion for the simply typed λ-calculus. The relation of convertibility up to α and β is written as $=_\beta$, and if η is added, is written as $=_{\beta\eta}$. We say that a λ-term is in β-normal form if it contains no beta redexes, that is, subformulas of the form $(\lambda x\ t)s$. A λ-term is in $\beta\eta$-long form if it is of the form

$$\lambda x_1 \ldots \lambda x_n (h t_1 \ldots t_m) \qquad (n, m \geq 0)$$

where h, called the head of the term, is either a constant or a variable, where the expression $ht_1 \ldots t_m$ is of primitive type, and where each term $t_1, \ldots, t_m$ are also in $\beta\eta$-long form. All λ-terms $\beta\eta$-convert to a term in $\beta\eta$-long form, unique up to α-conversion. See [HS86] for a fuller discussion of these basic properties of the simply typed λ-calculus.

Let Σ be a signature. A term is a Σ*-term* if all of its free variables and nonlogical constants are members of Σ. Similarly, a formula is a Σ*-formula* if all of its free variables and nonlogical

$$\frac{\Sigma\ ;\ B, C, \mathcal{P}\ \longrightarrow\ C}{\Sigma\ ;\ B \wedge C, \mathcal{P}\ \longrightarrow\ C}\ \wedge\text{-L} \qquad \frac{\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B \qquad \Sigma\ ;\ \mathcal{P}\ \longrightarrow\ C}{\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B \wedge C}\ \wedge\text{-R}$$

$$\frac{\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B \qquad \Sigma\ ;\ C, \mathcal{P}\ \longrightarrow\ A}{\Sigma\ ;\ B \supset C, \mathcal{P}\ \longrightarrow\ A}\ \supset\text{-L} \qquad \frac{\Sigma\ ;\ B, \mathcal{P}\ \longrightarrow\ C}{\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B \supset C}\ \supset\text{-R}$$

$$\frac{\Sigma\ ;\ [t/x]B, \mathcal{P}\ \longrightarrow\ C}{\Sigma\ ;\ \forall_\tau x\, B, \mathcal{P}\ \longrightarrow\ C}\ \forall\text{-L} \qquad \frac{\Sigma \cup \{c\colon\tau\}\ ;\ \mathcal{P}\ \longrightarrow\ [c/x]B}{\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ \forall_\tau x\, B}\ \forall\text{-R}$$

Figure 1: Left and right introduction rules for hh^ω

constants are members of Σ. A formula is either *atomic* or *non-atomic*. An atomic Σ-formula is of the form $(Pt_1 \ldots t_n)$, where $n \geq 0$, P is given type $\tau_1 \to \cdots \to \tau_n \to o$ by Σ, and $t_1, \ldots, t_n$ are terms of the types $\tau_1, \ldots, \tau_n$, respectively. The predicate constant P is the *head* of this atomic formula. Non-atomic formulas are of the form $\top$, $B_1 \wedge B_2$, $B_1 \supset B_2$, or $\forall_\tau x\ B$, where B, B_1, and B_2 are formulas and τ is a type not containing o.

The logic we have just presented is very closely related to two logic programming extensions that have been studied elsewhere [MNPS]. *First-order hereditary Harrop* formulas (*fohh*) have been studied as an extension to first-order Horn clauses as a basis for logic programming. Similarly *higher-order hereditary Harrop* formulas (*hohh*) are a generalization of *fohh* that permits some forms of predicate quantification. Because our meta-language is neither higher-order, since it lacks predicate quantification, nor first-order, since it contains quantification at all function types, we shall simply call it hh^ω. The set of hh^ω formulas in which quantification only up to order n is used will be labeled as hh^n.

Provability for hh^ω can be given in terms of sequent calculus proofs. A *sequent* is a triple $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B$, where Σ is a signature, B is a Σ-formula, and $\mathcal{P}$ is a finite (possibly empty) sets of Σ-formulas. The set $\mathcal{P}$ is this sequent's *antecedent* and B is its *succedent*. The expression $B, \mathcal{P}$ denotes the set $\mathcal{P} \cup \{B\}$; this notation is used even if $B \in \mathcal{P}$. The inference rules for sequents are presented in Figure 1. The following provisos are also attached to the two inference rules for quantifier introduction: in $\forall$-R the constant c is not in Σ, and in $\forall$-L t is a Σ-term of type τ.

A *proof* of the sequent $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B$ is a finite tree constructed using these inference rules such that the root is labeled with $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B$ and the leaves are labeled with *initial sequents*, that is, sequents $\Sigma'\ ;\ \mathcal{P}'\ \longrightarrow\ B'$ such that either B' is $\top$ or $B' \in \mathcal{P}'$. The non-terminals in such a tree are instances of the inference figures in Figure 1. Since we do not have an inference figure for $\beta\eta$-conversion, we shall assume that in building a proof, two formulas are equal if they are $\beta\eta$-convertible. If the sequent $\Sigma\ ;\ \mathcal{P}\ \longrightarrow\ B$ has a sequent proof then we write $\Sigma; \mathcal{P} \vdash_I B$ and say that B is provable from Σ and $\mathcal{P}$. We shall need only one proof-theoretic result concerning the meta-logic hh^ω. To state it, we require the following definition.

Definition 2.1 Let Σ be a signature and let $\mathcal{P}$ be a finite set of Σ-formulas. The expression $|\mathcal{P}|_\Sigma$ denotes the smallest set of pairs $\langle \mathcal{G}, D \rangle$ of finite sets of Σ-formulas $\mathcal{G}$ and Σ-formula D, such that

- If $D \in \mathcal{P}$ then $\langle \emptyset, D \rangle \in |\mathcal{P}|_\Sigma$.
- If $\langle \mathcal{G}, D_1 \wedge D_2 \rangle \in |\mathcal{P}|_\Sigma$ then $\langle \mathcal{G}, D_1 \rangle \in |\mathcal{P}|_\Sigma$ and $\langle \mathcal{G}, D_2 \rangle \in |\mathcal{P}|_\Sigma$.

- If $\langle \mathcal{G}, \forall_\tau x D\rangle \in |\mathcal{P}|_\Sigma$ then $\langle \mathcal{G}, [t/x]D\rangle \in |\mathcal{P}|_\Sigma$ for all Σ-terms t of type τ.
- If $\langle \mathcal{G}, G \supset D\rangle \in |\mathcal{P}|_\Sigma$ then $\langle \mathcal{G} \cup \{G\}, D\rangle \in |\mathcal{P}|_\Sigma$.

Theorem 2.2 A non-deterministic search procedure for hh^ω can be organized using the following four search primitives.

AND: $B_1 \wedge B_2$ is provable from Σ and $\mathcal{P}$ if and only if both B_1 and B_2 are provable from Σ and $\mathcal{P}$.

GENERIC: $\forall_\tau x B$ is provable from Σ and $\mathcal{P}$ if and only if $[c/x]B$ is provable from $\Sigma \cup \{c : \tau\}$ and $\mathcal{P}$ for any constant $c : \tau$ not in Σ.

AUGMENT: $B_1 \supset B_2$ is provable from Σ and $\mathcal{P}$ if and only if B_2 is provable from Σ and $\mathcal{P} \cup \{B_1\}$.

BACKCHAIN: The atomic formula A is provable from Σ and $\mathcal{P}$ if and only if there is a pair $\langle \mathcal{G}, A\rangle \in |\mathcal{P}|_\Sigma$ so that for every $G \in \mathcal{G}$, G is provable from Σ and $\mathcal{P}$.

These formal results are closely related to the notion of *expanded normal form* for natural deduction proofs [Pra71] which was used by Paulson in [Pau89] to establish the correctness of a specification of first-order logic in hh^ω. This theorem will similarly play a central role in proving the correctness of our representation of LF in hh^ω.

3 The Logical Framework

There are three levels of terms in the LF type theory: objects (often called just terms), types and families of types, and kinds. We assume two given denumerable sets of variables, one for object-level variables and the other for type family-level variables. The syntax of LF is given by the following classes of objects.

$$
\begin{array}{rcl}
K & := & \mathsf{Type} \mid \Pi x : A.K \\
A & := & x \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\
M & := & x \mid \lambda x : A.M \mid MN \\
\Gamma & := & \langle\rangle \mid \Gamma, x : K \mid \Gamma, x : A
\end{array}
$$

Here M and N range over expressions for objects, A and B over types and families of types, K over kinds, x over variables, and Γ over contexts. The empty context is denoted by $\langle\rangle$. We will use P and Q to range over arbitrary objects, types, type families, or kinds. We write $A \to P$ for $\Pi x : A.P$ when x does not occur in type or kind P. We will say that a type or type family of the form $xN_1 \dots N_n$ where $n \geq 0$ and x is a type family-level variable is a *flat type*.

Terms that differ only in the names of variables bound by λ or Π are identified. If x is an object-level variable and N is an object then $[N/x]$ denotes the operation of substituting N for all free occurrences of x, systematically changing bound variables in order to avoid variable capture. The expression $[N_1/x_1, \dots, N_n/x_n]$ will denote the simultaneous substitution of the terms $N_1, \dots, N_n$ for distinct variables $x_1, \dots, x_n$, respectively.

The notion of β-conversion at the level of objects, types, type families, and kinds can be defined in the obvious way using the usual rule for β-reduction at the level of both objects and type families: $(\lambda x : A.P)N \to_\beta [N/x]P$ where P is either an object or type/type family. The relation of convertibility up to β is written as $=_\beta$, just as it is at the meta-level. All well-typed LF terms are strongly normalizing [HHP87]. We write P^β to denote the normal form of term P. We present a version of the LF proof system that constructs only terms in canonical form, a notion which corresponds to $\beta\eta$-long forms in the simply typed λ-calculus. Several definitions from [14] are required to establish this notion. We define the *arity* of a type or kind to be the number of Πs in the prefix of its normal form. The arity of a variable with respect to a context is the arity of its type in that context. The arity of a bound variable occurrence in a term is the arity of the type label attached to its binding occurrence. An occurrence of a variable x in a term is *fully applied* with respect to a context if it occurs in a subterm of the form $xM_1 \dots M_n$, where n is the arity of x. A term P is *canonical* with respect to a context Γ if P is in β-normal form and every variable occurrence in P is fully applied with respect to Γ. We say that a context Γ is in canonical form if for every item $x : P$ in Γ, P is in canonical form with respect to Γ. Flat types of the form $xN_1 \dots N_n$ such that x is fully applied will be called *base types*.

The following three kinds of *assertions* are derivable in the LF type theory.

$$\begin{array}{ll} \Gamma \vdash K \text{ kind} & (K \text{ is a kind in } \Gamma) \\ \Gamma \vdash A : K & (A \text{ has kind } K \text{ in } \Gamma) \\ \Gamma \vdash M : A & (M \text{ has type } A \text{ in } \Gamma) \end{array}$$

We write $\Gamma \vdash \alpha$ for an arbitrary assertion, where α is called an LF *judgment*. For the special form $\Gamma \vdash A : \mathsf{Type}$ of the second type of assertion, we also say A is a type in Γ. A context $x_1 : P_1 \dots, x_n : P_n$ is said to be *valid* if $x_1, \dots, x_n$ are distinct variables and for $i = 1, \dots, n$, P_i is either a type or kind in context $x_1 : P_1, \dots, x_{i-1} : P_{i-1}$. In deriving one of the above assertions, we always assume that we start with a valid context Γ. We sometimes refer to the context in such an assertion as a *signature*.[1] Generally, a signature is a set of variables paired with types or kinds that specify an object logic and its inference rules.

The inference rules of LF are given in Figure 2. In (APP-OBJ) B must be a base type, and in (APP-OBJ) and (APP-FAM) n is the arity of x. In (PI-KIND), (PI-FAM), (ABS-FAM), and (ABS-OBJ), we assume that the variable x does not occur in Γ, and in (APP-FAM) and (APP-OBJ) we assume that the variables $x_1, \dots, x_n$ do not occur free in $N_1, \dots, N_n$. Note that bound variables can always be renamed to meet these restrictions.

The main difference between this presentation and the usual presentation of the LF proof system are the (APP-FAM) and (APP-OBJ) rules. The rules in the form presented here are those needed to preserve the invariant that all objects, types, type families, kinds, and contexts in derivable judgments are in canonical form. To see why, first note that no new β-redexes are introduced in the conclusion of these rules. The application introduced in the left term of the judgment in the conclusion is always a variable applied to zero or more terms, while the right term is always β-normal. Second, note that the signature item x of arity n is applied in the conclusion to n terms and thus this occurrence of x is fully applied. Hence, as long as $N_1, \dots, N_n$ are canonical, so is $xN_1 \dots N_n$. In the (APP-OBJ) rule, the fact that the type $([N_1/x_1, \dots, N_n/x_n]B)^\beta$ is canonical follows from the fact that for any object, type, type family, or kind P and any object M, if P and M are canonical, then so is $([M/x]P)^\beta$. Based on these observations, derivable assertions can be characterized more formally as follows: if Γ is a valid context, then Γ is canonical, and if $\Gamma \vdash \alpha$ is

[1] Other presentations of LF such as [HHP87] separate the notions of context and signature. We unify them here for simplicity of presentation.

$$\Gamma \vdash \mathsf{Type}\ \mathrm{kind} \quad \text{(TYPE-KIND)}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash K\ \mathrm{kind}}{\Gamma \vdash \Pi x{:}A.K\ \mathrm{kind}}\ \text{(PI-KIND)}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi x{:}A.B : \mathsf{Type}}\ \text{(PI-FAM)}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : K}{\Gamma \vdash \lambda x{:}A.B : \Pi x{:}A.K}\ \text{(ABS-FAM)}$$

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}\ \text{(ABS-OBJ)}$$

$$\frac{\begin{array}{l} x : \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\mathsf{Type} \in \Gamma \\ \Gamma \vdash N_1 : A_1 \\ \Gamma \vdash N_2 : ([N_1/x_1]A_2)^\beta \\ \vdots \\ \Gamma \vdash N_n : ([N_1/x_1, \ldots, N_{n-1}/x_{n-1}]A_n)^\beta \end{array}}{\Gamma \vdash xN_1 \ldots N_n : \mathsf{Type}}\ \text{(APP-FAM)}$$

$$\frac{\begin{array}{l} x : \Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B \in \Gamma \\ \Gamma \vdash N_1 : A_1 \\ \Gamma \vdash N_2 : ([N_1/x_1]A_2)^\beta \\ \vdots \\ \Gamma \vdash N_n : ([N_1/x_1, \ldots, N_{n-1}/x_{n-1}]A_n)^\beta \end{array}}{\Gamma \vdash xN_1 \ldots N_n : ([N_1/x_1, \ldots, N_n/x_n]B)^\beta}\ \text{(APP-OBJ)}$$

Figure 2: The Logical Framework

derivable with respect to valid context Γ, then the terms in α are canonical with respect to Γ, and α has one of the following forms.

1. $\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\mathsf{Type}$ kind where $n \geq 0$.

2. $(\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.\Pi z_1{:}B_1 \ldots \Pi z_m{:}B_m.C) : (\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.\mathsf{Type})$ where $n, m \geq 0$ and C is a base type.

3. $(\lambda x_1{:}A_1 \ldots \lambda x_n{:}A_n.N) : (\Pi x_1{:}A_1 \ldots \Pi x_n{:}A_n.B)$ where $n \geq 0$, N is not an abstraction, and B is a base type.

Note that proving an assertion of the form given by (2) or (3), respectively, in valid context Γ, is equivalent to proving $\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n \vdash \Pi z_1{:}B_1 \ldots \Pi z_m{:}B_m.C : \mathsf{Type}$ or $\Gamma, x_1 : A_1, \ldots, x_n : A_n \vdash N : B$, respectively, in valid context $\Gamma, x_1{:}A_1, \ldots, x_n{:}A_n$. In the first version of the translation given in the next section, we will assume that assertions have the latter form, *i.e.*, that there are no leading abstractions in the term on the left in a judgment.

4 Translating LF Assertions to hh^ω Formulas

In this section we present the translation of LF assertions to formulas in hh^ω. This translation will require an encoding of LF terms as simply typed λ-terms. We begin by presenting this encoding. We then present the translation, which given an LF assertion, $\Gamma \vdash \alpha$ where Γ is a valid context, translates Γ to a set of hh^ω formulas and α to a formula to be proved from this set of formulas. We then illustrate how to extend the translation to obtain a formula whose proof (from no assumptions) verifies that Γ is a valid context before proving that α holds within the context Γ.

Since both LF and the meta-language have types and terms, to avoid confusion we will refer to types and terms of the meta-language as *meta-types* and *meta-terms*. We only define the encoding of LF terms as simply typed λ-terms for LF objects and flat types/type families since this is all that is required by the translation. We introduce two base types, tm and ty, at the meta-level for these two classes of LF terms. First, to encode object-level variables, we define the function Φ that maps LF types and kinds to meta-types containing only occurrences of tm and ty.

$$\begin{array}{rcl} \Phi(\Pi x\!:\!A.P) & := & \Phi(A) \rightarrow \Phi(P) \\ \Phi(\mathsf{Type}) & := & ty \\ \Phi(A) & := & tm \quad \text{when } A \text{ is a flat type} \end{array}$$

Using this function, an LF variable of kind or type P is mapped to a meta-variable of type $\Phi(P)$. These meta-types encode the "syntactic structure" of the corresponding LF dependent type or kind. Information about dependencies is lost in this mapping, but as we will see later, this information is retained in a different form in performing the general translation. We will assume a fixed mapping from LF variables to meta-variables of the corresponding type. For readability in our presentation, this mapping will be implicit. A variable x will represent both an LF variable with kind or type P and a meta-variable of the corresponding syntactic type $\Phi(P)$. It will always be clear from context which is meant. Note that for type or kind P and object N, $\Phi(P) = \Phi(([N/x]P)^\beta)$.

We denote the encoding of term or flat type P as $\langle\!\langle P \rangle\!\rangle$. The full encoding is defined below.

$$\begin{array}{rcl} \langle\!\langle x \rangle\!\rangle & := & x \\ \langle\!\langle \lambda x\!:\!A.M \rangle\!\rangle & := & \lambda x\!:\!\Phi(A).\langle\!\langle M \rangle\!\rangle \\ \langle\!\langle MN \rangle\!\rangle & := & \langle\!\langle M \rangle\!\rangle\ \langle\!\langle N \rangle\!\rangle \\ \langle\!\langle AM \rangle\!\rangle & := & \langle\!\langle A \rangle\!\rangle\ \langle\!\langle M \rangle\!\rangle \end{array}$$

Note that the encoding maps abstraction in LF objects directly to abstraction at the meta-level, and that both application of objects to objects and application of type families to objects are mapped directly to application at the meta-level. The difference at the meta-level is that the former application will be a meta-term with target type tm while the latter application will be a meta-term with target type ty.

It is easy to see that for object or type family P having, respectively, type or kind Q, $\langle\!\langle P \rangle\!\rangle$ is a meta-term of meta-type $\Phi(Q)$. The following two properties also hold for this encoding.

Lemma 4.1 Let P be an LF object or flat type, and N an LF object. Then

$$[\langle\!\langle N \rangle\!\rangle/x]\langle\!\langle P \rangle\!\rangle = \langle\!\langle [N/x]P \rangle\!\rangle.$$

Lemma 4.2 Let P and Q be two LF objects or flat types. If $P =_\beta Q$, then $\langle\!\langle P \rangle\!\rangle =_\beta \langle\!\langle Q \rangle\!\rangle$.

We are now ready to define the translation. Two predicates will appear in the atomic hh^ω formulas resulting from the translation: *hastype* of type $tm \to ty \to o$ and *istype* of type $ty \to o$. We will name the signature containing these two predicates Σ_{LF}. We denote the translation of the context item or judgment α as $[\![\alpha]\!]$. The full translation is defined in Figure 3. It is a partial function since it is defined by cases and undefined when no case applies. It will in fact always be defined on contexts and judgments in provable LF assertions. In proving properties of the translation, we will only consider canonical judgments and context items. Note that in a canonical context item $x\!:\!P$, the variable x is not necessarily canonical since it may not be fully applied. Such judgments with non-canonical terms on the left are handled by the first and third rules in Figure 3. This translation maps occurrences of Π-abstraction in LF types and kinds directly to instances of

$$
\begin{array}{rcl}
[\![M : \Pi x\!:\!A.B]\!] & := & \forall_{\Phi(A)} x \left([\![x\!:\!A]\!] \supset [\![Mx : B]\!]\right) \\
[\![M : A]\!] & := & \mathit{hastype}\ \langle\!\langle M \rangle\!\rangle\ \langle\!\langle A \rangle\!\rangle \quad \text{where } A \text{ is a base type.} \\
[\![B : \Pi x\!:\!A.K]\!] & := & \forall_{\Phi(A)} x \left([\![x\!:\!A]\!] \supset [\![Bx : K]\!]\right) \\
[\![A : \mathsf{Type}]\!] & := & \mathit{istype}\ \langle\!\langle A \rangle\!\rangle \quad \text{where } A \text{ is a base type.} \\
[\![\Pi x\!:\!A.B : \mathsf{Type}]\!] & := & [\![A : \mathsf{Type}]\!] \wedge \forall_{\Phi(A)} x \left([\![x\!:\!A]\!] \supset [\![B : \mathsf{Type}]\!]\right) \\
[\![\mathsf{Type}\ \text{kind}]\!] & := & \top \\
[\![\Pi x\!:\!A.K\ \text{kind}]\!] & := & [\![A : \mathsf{Type}]\!] \wedge \forall_{\Phi(A)} x \left([\![x\!:\!A]\!] \supset [\![K\ \text{kind}]\!]\right)
\end{array}
$$

Figure 3: Translation of LF Judgments to hh^ω Formulas

universal quantification and implication in hh^ω formulas. In all of the clauses in the definition that contain a pattern with a Π-type or kind, the variable bound by Π is mapped to a variable at the meta-level bound by universal quantification. Then, in the resulting implication, the left hand side asserts the fact that the bound variable has a certain type, while the right hand side contains the translation of the body of the type or kind which may contain occurrences of this bound variable. The base cases occur when there is no leading Π in the type or kind, resulting in atomic formulas for the *hastype* and *istype* predicates, or simply $\top$ in the case when the judgment is Type kind.

To illustrate this translation, we consider an example from an LF signature specifying natural deduction for first-order logic. The following declaration introduces the constant for universal quantification and gives it a type: $\forall^* : (i \to \mathit{form}) \to \mathit{form}$. (We write $\forall^*$ for universal quantification at the object level to distinguish it from universal quantification in hh^ω.) To make all bound variables explicit, we expand the above type to its unabbreviated form: $\Pi A : (\Pi y : i.\mathit{form}).\mathit{form}$. Note that by applying Φ to the above type, we get $(tm \to tm) \to tm$ as the type of $\forall^*$ at the meta-level. The translation of this signature item is as follows.

$$
\begin{array}{l}
[\![\forall^* \ :\ \Pi A\!:\!(\Pi y\!:\!i.\mathit{form}).\mathit{form}]\!] \equiv \\
\qquad \forall_{tm\to tm} A \left(\forall_{tm} y \left([\![y : i]\!] \supset [\![Ay : \mathit{form}]\!]\right) \supset [\![\forall^* A : \mathit{form}]\!]\right) \equiv \\
\qquad\qquad \forall_{tm\to tm} A\ (\forall_{tm} y\ ((\mathit{hastype}\ y\ i) \supset (\mathit{hastype}\ (Ay)\ \mathit{form})) \supset (\mathit{hastype}\ (\forall^* A)\ \mathit{form}))
\end{array}
$$

This formula provides the following description of the information contained in the above dependent type: for any A, if for arbitrary y of type i, Ay is a formula, then $\forall^* A$ is a formula.

We will show in the next section that if Γ is a valid canonical context and α a canonical judgment where the term on the left in α is not an abstraction, then $\Gamma \vdash \alpha$ is provable in LF iff $[\![\alpha]\!]$ is provable from the set of formulas $[\![\Gamma]\!]$. (Here $[\![\Gamma]\!]$ denotes the set of formulas obtained by translating separately each item in Γ.) We now illustrate how to extend the translation to obtain a formula whose proof verifies that Γ is a valid context before proving that α holds within the context Γ.

Proving that a context $x_1 : P_1, \ldots, x_n : P_n$ is valid in LF corresponds in hh^ω to proving, for $i = 1, \ldots, n$, either $[\![P_i : \mathsf{Type}]\!]$ or $[\![P_i \text{ kind}]\!]$ from $[\![x_1 : P_1, \ldots, x_{i-1} : P_{i-1}]\!]$. The translation in Figure 4, for an arbitrary assertion $\Gamma \vdash \alpha$, maps the pair $(\Gamma; \alpha)$ to a single formula containing subformulas whose proofs will in fact insure that each context item is valid with respect to the context items before it. We also remove the restriction that the term on the left in α cannot be an abstraction. Variables bound by abstraction at the top level are treated as additional context items. The translation of such a pair is denoted $[\![\Gamma; \alpha]\!]^*$. The first two clauses of this translation

$$
\begin{array}{rcl}
[\![x : A, \Gamma; \alpha]\!]^* & := & [\![A : \mathsf{Type}]\!] \wedge \forall_{\Phi(A)} x \left([\![x : A]\!] \supset [\![\Gamma; \alpha]\!]^*\right) \\
[\![x : K, \Gamma; \alpha]\!]^* & := & [\![K \text{ kind}]\!] \wedge \forall_{\Phi(K)} x \left([\![x : K]\!] \supset [\![\Gamma; \alpha]\!]^*\right) \\
[\![\langle\rangle; \lambda x : A.M : \Pi x : A.B]\!]^* & := & [\![A : \mathsf{Type}]\!] \wedge \forall_{\Phi(A)} x \left([\![x : A]\!] \supset [\![\langle\rangle; M : B]\!]^*\right) \\
[\![\langle\rangle; \lambda x : A.B : \Pi x : A.K]\!]^* & := & [\![A : \mathsf{Type}]\!] \wedge \forall_{\Phi(A)} x \left([\![x : A]\!] \supset [\![\langle\rangle; B : K]\!]^*\right) \\
[\![\langle\rangle; \alpha]\!]^* & := & [\![\alpha]\!] \quad \text{where the left term in } \alpha \text{ is not an abstraction.}
\end{array}
$$

Figure 4: Translation of LF Assertions

map each context item to a conjunctive formula where the first conjunct verifies that the type or kind is valid (using the previous translation), and the second conjunct is a universally quantified implication where the left hand side asserts the fact that the context item has the corresponding type (again using the previous translation), and the right side contains the translation of the pair consisting of the remaining context items and judgment. The third and fourth clauses handle the cases when the term on the left in a judgment is an abstraction. The last clause in the translation is for the base case: when the context is empty and no further abstractions remain at the head of the judgment. Then the previously defined translation is used. Thus, a proof of a formula obtained from translating an arbitrary assertion $\Gamma \vdash \alpha$ with respect to an initially empty set of assumptions verifies that each context item in Γ, and each variable bound by λ-abstraction in α is valid with respect to those items that appear before it, and then proves that the judgment holds within the entire context. The correctness of this translation will follow easily from the correctness of the previous translation.

5 Correctness of Translation

We consider the correctness of the translation with respect to a slightly modified LF. Our modified system replaces the (ABS-FAM) and (ABS-OBJ) rules with the following two rules.

$$\frac{\Gamma, x\!:\!A \vdash B : K}{\Gamma \vdash \lambda x\!:\!A.B : \Pi x\!:\!A.K}\ (\text{ABS-FAM}') \qquad \frac{\Gamma, x\!:\!A \vdash M : B}{\Gamma \vdash \lambda x\!:\!A.M : \Pi x\!:\!A.B}\ (\text{ABS-OBJ}')$$

These rules are the same as presented earlier except that the left premise is omitted. We call this system LF$'$. It can be shown that for valid context Γ, an LF assertion $\Gamma \vdash \alpha$ is provable in LF$'$ if and only if it is provable in LF, provided that there is no leading abstraction in the term on the left in α. This result relies on the fact that in a proof in the modified system, if an application of (ABS-FAM$'$) whose conclusion is $\Gamma \vdash \lambda x\!:\!A.B : \Pi x\!:\!A.K$ or an application of (ABS-OBJ$'$) whose conclusion is $\Gamma \vdash \lambda x\!:\!A.M : \Pi x\!:\!A.B$ occurs above an application of (APP-FAM) or (APP-OBJ), it is always the case that $\Gamma \vdash A : \mathsf{Type}$ is provable, and thus the left premise is redundant. The proof of this fact relies on a transitivity result for LF$'$ similar to the one stated in [HHP87].

To prove the correctness of the translation, we prove a stronger statement from which the correctness of $[\![\ ]\!]$ and $[\![\ ]\!]^*$ will follow as corollaries. This stronger statement will talk about the provability of LF assertions of the form $\Gamma \vdash \alpha$ in LF$'$ even in the case when Γ is not a valid context. We also relax the requirement on α. The left term in α can be any object, type, type family, or kind, including one with a leading abstraction. To handle these cases in proving the correctness of $[\![\ ]\!]$, we must add the following two rules to the translation.

$$\begin{array}{lcl} [\![\lambda x\!:\!A.M : \Pi x\!:\!A.B]\!] & := & \forall_{\Phi(A)} x \left([\![x : A]\!] \supset [\![M : B]\!]\right) \\ [\![\lambda x\!:\!A.B : \Pi x\!:\!A.K]\!] & := & \forall_{\Phi(A)} x \left([\![x : A]\!] \supset [\![B : K]\!]\right) \end{array}$$

We must then also add the restriction that the first and third rules in Figure 3 are only applicable when M and B, respectively, are not abstractions.

One final lemma is needed to prove the correctness of the translation. This lemma applies to the translation extended with the above two rules. In Section 4, we stated that substitution commutes with the encoding operation (Lemma 4.1). We extend this result to the translation operation on judgments which translate to provable hh^ω formulas. In particular, the lemma below states that substitution and β-normalization commute with the translation operation on provable hh^ω sequents. We will write $\Phi(\Gamma)$ to denote the set of meta-variables paired with their types obtained by mapping, for each signature item $x\!:\!P$ in Γ, the variable x to the corresponding meta-variable and P to $\Phi(P)$.

Lemma 5.1 Let $\Gamma, x_1 : A_1, \ldots, x_n : A_n, x : A$ $(n \geq 0)$ be a canonical context (whose variables are distinct). Let $N_1, \ldots, N_n, N$ be canonical objects with respect to Γ, and let Σ be the signature $\Sigma_{LF} \cup \Phi(\Gamma)$. Then $\Sigma; [\![\Gamma]\!] \vdash_I [\![N : ([N_1/x_1, \ldots, N_n/x_n]A)^\beta]\!]$ if and only if

$$\Sigma; [\![\Gamma]\!] \vdash_I [\langle\!\langle N_1 \rangle\!\rangle/x_1, \ldots, \langle\!\langle N_n \rangle\!\rangle/x_n, \langle\!\langle N \rangle\!\rangle/x][\![x : A]\!].$$

Theorem 5.2 (Correctness of Translation) Let Γ be an arbitrary context (such that the variables in Γ are distinct), and let α be an arbitrary canonical judgment with respect to Γ. Let Σ be $\Sigma_{LF} \cup \Phi(\Gamma)$. Then $\Gamma \vdash \alpha$ is provable in LF$'$ if and only if $\Sigma; [\![\Gamma]\!] \vdash_I [\![\alpha]\!]$ holds.

Proof Sketch: The proof of this theorem is constructive, *i.e.*, it provides a method for constructing an hh^ω proof from an LF$'$ proof, and vice versa. The forward direction is proved by induction on the height of an LF$'$ proof of the assertion $\Gamma \vdash \alpha$. For the PI and ABS rules, we can apply the induction hypothesis directly to the premises to obtain provable sequents to which we apply $\supset$-R, $\forall$-R, and $\wedge$-R for the PI rules, and just $\supset$-R and $\forall$-R for the ABS rules to obtain the desired result.

For the APP rules, we know the context item in the application of these rules corresponds to a formula in $[\![\Gamma]\!]$. To this formula, we can apply $\forall$-L followed by $\supset$-L n times in a backward direction. Each of the left premises of $\supset$-L can be shown to be provable since they are the result of applying the induction hypothesis followed by Lemma 5.1 to each of the latter n premises of the APP rule. Using Lemmas 4.1 and 4.2, the formula introduced on the left in the right premise of the topmost application of $\supset$-L can be shown to be β-convertible to the formula in the succedent (the translation of the judgment in the conclusion of the APP rule). Thus this premise is an axiom.

The proof of the backward direction is by induction on the structure of the term on the left in α, and is similar to the proof of the forward direction. The regularity of the proofs in hh^ω described in Theorem 2.2 is required here. For example, the proof of the case when the term on the left is an application uses the backchain search operation. ■

Corollary 5.3 (Correctness of $[\![\ ]\!]$) Let Γ be a valid context and α a canonical judgment such that the term on the left is not an abstraction. Let Σ be $\Sigma_{LF} \cup \Phi(\Gamma)$. Then $\Gamma \vdash \alpha$ is provable in LF$'$ if and only if $\Sigma; [\![\Gamma]\!] \vdash_I [\![\alpha]\!]$ holds.

Corollary 5.4 (Correctness of $[\![\ ]\!]^*$) Let Γ be a canonical context (such that the variables in Γ are distinct), and α a canonical judgment. Then Γ is a valid context and $\Gamma \vdash \alpha$ is provable in LF$'$ if and only if $\Sigma_{LF}; \emptyset \vdash_I [\![\Gamma; \alpha]\!]^*$ holds.

6 Examples

In this section, we provide some further examples to illustrate the correspondence between LF signature items and judgments and the hh^ω formulas that they map to. Note that in general, formulas obtained by translating signature items have the form on the left below, but can be rewritten to have the form on the right:

$$\forall_{\tau_1} X_1\ (G_1 \supset \ldots \forall_{\tau_n} X_n\ (G_n \supset D) \ldots) \qquad\qquad \forall_{\tau_1} X_1 \ldots \forall_{\tau_n} X_n\ (G_1 \wedge \cdots \wedge G_n \supset D)$$

where $n \geq 0$, $\tau_1, \ldots, \tau_n$ are types, $X_1, \ldots, X_n$ are variables, $G_1, \ldots, G_n, D$ are hh^ω formulas. (Here we assume that for $i = 1, \ldots, n$, $X_{i+1}, \ldots, X_n$ do not appear free in G_i). For readability, we will write hh^ω formulas in the examples in this section simply as $G_1 \wedge \cdots \wedge G_n \supset D$ (or just D when $n = 0$), and assume implicit universal quantification over all free variables written as capital letters. Type subscripts for these universal quantifiers can always be inferred from context.

We begin by demonstrating the translation of signature items specifying natural deduction inference rules for the $\wedge^*$, $\forall^*$, and $\supset^*$ object-level connectives. The fragment of an LF signature specifying natural deduction for the first-order logic that we are concerned with is the following.

$$\begin{aligned}
i &: \mathsf{Type} \\
\mathit{form} &: \mathsf{Type} \\
\mathit{true} &: \mathit{form} \to \mathsf{Type} \\
\wedge^* &: \mathit{form} \to \mathit{form} \to \mathit{form} \\
\supset^* &: \mathit{form} \to \mathit{form} \to \mathit{form} \\
\forall^* &: (i \to \mathit{form}) \to \mathit{form} \\
\wedge^*\text{-I} &: \Pi A{:}\mathit{form}.\Pi B{:}\mathit{form}.\mathit{true}(A) \to \mathit{true}(B) \to \mathit{true}(A \wedge^* B) \\
\supset^*\text{-I} &: \Pi A{:}\mathit{form}.\Pi B{:}\mathit{form}.(\mathit{true}(A) \to \mathit{true}(B)) \to \mathit{true}(A \supset^* B) \\
\forall^*\text{-I} &: \Pi A{:}i \to \mathit{form}.(\Pi y{:}i.\mathit{true}(Ay)) \to \mathit{true}(\forall^* A)
\end{aligned}$$

The signature item *true* is a function that maps formulas to types. LF objects of type $\mathit{true}(A)$ represent proofs of formula A. First, consider the $\wedge^*$-introduction rule specified by $\wedge^*$-I and its type. Its translation is the following formula.

$$\begin{aligned}
&(\mathit{hastype}\ A\ \mathit{form}) \wedge (\mathit{hastype}\ B\ \mathit{form}) \wedge (\mathit{hastype}\ P\ (\mathit{true}\ A)) \wedge \\
&\quad (\mathit{hastype}\ Q\ (\mathit{true}\ B)) \supset (\mathit{hastype}\ (\wedge^*\text{-I}\ A\ B\ P\ Q)\ (\mathit{true}\ A \wedge^* B))
\end{aligned}$$

This formula simply reads that if A and B have type *form*, P is a proof of A, and Q is a proof of B, then the term $(\wedge^*\text{-I}\ A\ B\ P\ Q)$ is a proof of the conjunction $A \wedge^* B$. The correspondence between this formula and the LF signature item is straightforward. We next consider a slightly more complex example; the translation of the $\forall^*$-I rule results in the following formula.

$$\begin{aligned}
&\forall y((\mathit{hastype}\ y\ i) \supset (\mathit{hastype}\ Ay\ \mathit{form})) \wedge \\
&\quad \forall y((\mathit{hastype}\ y\ i) \supset (\mathit{hastype}\ Py\ (\mathit{true}\ Ay))) \supset (\mathit{hastype}\ (\forall^*\text{-I}\ A\ P)\ (\mathit{true}\ \forall^* A))
\end{aligned}$$

This clause provides the following description of the information contained in the dependent type: if for arbitrary y of type i, Ay is a formula and Py is a proof of Ay, then the term $(\forall^*\text{-I}\ A\ P)$ is a proof of $\forall^* A$. Note that A and P at the meta-level are both functions having syntactic type $tm \to tm$. Here, A maps first-order terms to formulas just as it does at the object level, while P maps first-order terms to proofs. As a final inference rule example, consider the declaration for $\supset^*$-I, which translates to the following formula.

$$\begin{aligned}
&(\mathit{hastype}\ A\ \mathit{form}) \wedge (\mathit{hastype}\ B\ \mathit{form}) \wedge \\
&\quad \forall q((\mathit{hastype}\ q\ (\mathit{true}\ A)) \supset (\mathit{hastype}\ Pq\ (\mathit{true}\ B))) \supset \\
&\qquad (\mathit{hastype}\ (\supset^*\text{-I}\ A\ B\ P)\ (\mathit{true}\ A \supset^* B))
\end{aligned}$$

This formula reads: if A and B are formulas and P is a function which maps an arbitrary proof q of A to the proof Pq of B, then the term $(\supset^*\text{-I}\ A\ B\ P)$ is a proof of $A \supset^* B$. Note that P in this formula is a function which maps proofs to proofs.

An example of a canonical judgment that is provable in the LF signature for natural deduction is

$$\lambda A{:}\mathit{form}.{\supset^*}\text{-I}(A)(A)(\lambda x{:}\mathit{true}(A).x)\ :\ \Pi A{:}\mathit{form}.\mathit{true}(A \supset^* A).$$

Using the extended translation of Figure 4, we obtain the following formula:

$$(\mathit{istype}\ \mathit{form}) \wedge \forall A((\mathit{hastype}\ A\ \mathit{form}) \supset (\mathit{hastype}\ (\supset^*\text{-I}\ A\ A\ \lambda x.x)\ (\mathit{true}\ A \supset^* A)))$$

which is provable from the set of formulas obtained by translating the LF signature specifying natural deduction for first-order logic.

An LF signature specifying the reductions needed for proof normalization in natural deduction is given in [14]. As a final example, we illustrate the translation of the reduction rule for the case when an application of the introduction rule for $\supset^*$ is followed by the elimination rule for the same connective. The following signature items define the $\supset^*$-E rule, the *reduce* constant used to relate two proofs of the same formula, and the reduction rule for $\supset^*$.

$$\begin{aligned}
\supset^*\text{-E} &: \Pi A{:}form.\Pi B{:}form.true(A) \to true(A \supset^* B) \to true(B)\\
reduce &: \Pi A{:}form.true(A) \to true(A) \to \mathsf{Type}\\
\supset^*\text{-}red &: \Pi A{:}form.\Pi B{:}form.\Pi P{:}(true(A) \to true(B)).\Pi Q{:}true(A).\\
&\qquad reduce(B)(\supset^*\text{-E } A\ B\ (\supset^*\text{-I } A\ B\ P)\ Q)(PQ)
\end{aligned}$$

The signature item for $\supset^*$-*red* translates to the following formula.

$$\begin{aligned}
&(hastype\ A\ form) \wedge (hastype\ B\ form) \wedge\\
&\quad \forall q\ ((hastype\ q\ (true\ A)) \supset (hastype\ (Pq)\ (true\ B))) \wedge (hastype\ Q\ (true\ A)) \supset\\
&\qquad (hastype\ (\supset^*\text{-}red\ A\ B\ P\ Q)\ (reduce\ B\ (\supset^*\text{-E } A\ B\ (\supset^*\text{-I } A\ B\ P)\ Q)\ (PQ)))
\end{aligned}$$

This formula reads: if A and B are formulas and P is a function which maps an arbitrary proof q of A to the proof Pq of B, and Q is a proof of A, then $(\supset^*\text{-}red\ A\ B\ P\ Q)$ is a meta-proof of the fact that the natural deduction proof $(\supset^*\text{-E } A\ B\ (\supset^*\text{-I } A\ B\ P)\ Q)$ of B reduces to the proof PQ.

7 Conclusion

We have not yet considered the possibility of translating hh^ω formulas into LF. This translation is particularly simple. Let Σ be a signature for hh^ω and let $\mathcal{P}$ be a set of Σ-formulas. For each primitive type τ other than o in S, the corresponding LF judgment is $\tau : \mathsf{Type}$. For each non-predicate constant $c : \tau \in \Sigma$, the corresponding LF judgment is $c : \tau$. For each predicate constant $p : \tau_1 \to \cdots \to \tau_n \to o \in \Sigma$, the corresponding LF judgment is $p : \tau_1 \to \cdots \to \tau_n \to \mathsf{Type}$. Finally, let $D \in \mathcal{P}$ and let k be a new constant not used in the translation to this point. Then the corresponding LF judgment is $k : D'$ where D' is essentially D where $B_1 \supset B_2$ is written as $\Pi x{:}B_1.B_2$ and $\forall_\tau x\ B$ is written as $\Pi x{:}\tau.B$.

In the first author's dissertation [Fel89] an encoding of LF into just hh^2 was presented. Order 2 is all that is necessary if object-level applications are represented by meta-level constants. The proofs of the correctness of that encoding are very similar to those presented here.

Notice that the translation presented here works via recursion over the structure of types. Thus, this kind of translation will not work for the polymorphic λ-calculus or the Calculus of Constructions since they both contain quantification over types. Other techniques can be used, however, to encode provability of such λ-calculi into hh^ω. These involve coding the provability relation of those calculi directly into the meta-language [FM89].

Acknowledgements The authors would like to thank Robert Harper and Frank Pfenning for valuable discussions on the subject of this paper. We are also grateful to the reviewers of an earlier draft of this paper for their comments and corrections. Both authors have been supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018. The first author is currently supported in part by ESPRIT Basic Research Action 3245.

References

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language*. PhD thesis, University of Pennsylvania, August 1989.

[FM89] Amy Felty and Dale Miller. A meta language for type checking and inference: an extended abstract. 1989. Presented at the 1989 Workshop on Programming Logic, Bålstad, Sweden.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, June 1987.

[HHP89] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. 1989. Technical Report CMU-CS-89-173, to appear.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[MNPS] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. To appear in the Annals of Pure and Applied Logic.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[Pau89] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

[Pfe89] Frank Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.

[Pra71] Dag Prawitz. Ideas and results in proof theory. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 235–307, North-Holland, Amsterdam, 1971.

LOGIC vs LOGICS
A methodological panel

Jean Yves Girard

Université Paris 7 et INRIA

The discussion is open to all participants. The discussant tries to cover a wide sample of (controversial) issues, but does not claim to be neutral on these issues.

1 General considerations as to logic

1.1 Logic = formal systems

(analogy: mechanics = differential equations). Possible technical formulation: existence of a Gentzen-type formulation enjoying cut-elimination (symmetry intro/elim).

1.2 Logic = (modular) maintenance of (hidden) data

* classical: maintenance of truth values
* intuitionistic: maintenance of existential outputs (pb of unicity of this output ; ensured by either denotational semantics or Church-Rosser property)

1.3 Logic should be able to cope with formalisation of (rigourous) human reasoning

* point: rigourous = mathematical
* example: moves at chess very akwardly captured by classical logic ("frame axioms" etc.) but naturally handled by linear logic.
* a main issue: proof search
  automatic thm proving vs
  automatic thm checking vs
  assisted pf-search vs
  pb solving environment (Constable)
* the pb of backwards search (retro causality) seems to be central. Can we seriously think of a retro-implication ?

2 The myth of unity of logic

2.1 Logical frameworks

Idea to declare one's logic in a general neutral metalogical system. Danger of synchretism, equalisation by the bottom, not to speak of the multiplication of those frameworks.

However the idea of being able to declare new connectives (with some proviso, e.g. local cut-elim) is quite pleasant.

2.2 Refinements of sequent calculus

* main parameter = structural rules (might include new ones besides the standard WCE). Would it be possible to declare sequent in which formulas are

    - partially ordered (for retrocausality)
    - on some of them some structural rules are legal.

* criterion of success: translation of usual logical systems inside this refined sequent calculus ($\neq$ axiomatising).

3 A controversial issue: predicative vs impredicative

3.1 Does anybody still believe than impredicativity is dangerous ?

* crisis of foundations $\sim$ 1900

* personal opinion: “RINGARD”

3.2 Technical comparison

* impredicative (system F, Constructions) more modular

* predicative (Martin-Löf type theory) copes with subtle points (e.g. 1 step predecessor).

3.3 Predicativity and feasibility

* Predicative programs are (in theory) much shorter than impredicative ones, but still far from being feasible. In fact intuitionistic implication is a nasty impredicative construction. At the moment only bounded linear logic yields feasible (=polytime) bounds.

* dream of a two-level system

    - impredicative for coarse I/O specifications
    - (strongly) predicative for subtler (e.g. polytime) specifications

Models of partial inductive definitions

Lars Hallnäs

Department of Computer Sciences
Chalmers University of Technology and University of Göteborg
412 96 Göteborg, Sweden

1 Introduction

In [3] we gave an interpretation of a class of inductive definitions as partial definitions. The interpretation considered there was a *logical* one. We tried to interpret a definition in terms of the *local logic* generated by it. But if we want to be able to make finer intensional distinctions we have to consider a type of interpretations that preserves more of the intensional structure of the definition. The problem with this on the other hand is that the notion of structure involved is of a rather formal nature and perhaps not stable enough. The interpretation discussed here is a reduction to tuples of simple definitions where the structure is intuitively clear. We will roughly associate a tuple $(D_\star, D^\star, D_T)$ with each definition D, where $D_\star$ and $D^\star$ are simple definitions over a common universe U and D_T a *copy* of D in the form of a partition of U. We will try to view $(D_\star, D^\star, D_T)$ somehow as the primary object, a sort of model of D through which we can study the intensional structure of D. This type of interpretation is technically closely related to a Curry-Howard interpretation of D as a type system for simply typed lambda calculus with abstract data types (see [2, 3]). A lot of inspiration comes initially from Prawitz work on a general proof theory ([8]). But since we are looking for a reduction to simple monotone definitions abstraction and substitution will be treated in a non standard manner as *internal local* operations (cf. [4]). There are no external *syntactical* notions. All basic notions are given in terms of the structure of the simple definitions as such. In order to get as rich a structure as possible we also consider a kind of *completion* of the standard interpretation. What follows below is a preliminary version.

2 Simple definitions

Let U be a given universe of discourse. We call the objects in U *atoms* and the objects in $U \cup P(U) \cup \{\perp, \top\}$ *conditions*. We will use $a, b, p, q, r, \ldots$ to denote atoms and use A, B, C to denote conditions. By a *simple definition* over U we then understand a set D of equations

$$\begin{cases} a & = & A \\ & \vdots & \end{cases}$$

where a is an atom and A a condition. This notion of a simple definition correspond to Aczels interpretation of monotone inductive definitions in terms of *rule sets* in [1]. Let

$$\begin{array}{rcl} Dom(D) & = & \{a \mid (a = A) \in D \text{ for some } A\} \\ \hat{D} & = & D \cup \{a = \perp \mid a \notin Dom(D)\} \\ D(a) & = & \{A \mid (a = A) \in \hat{D}\} \end{array}$$

[1]This research has been carried out as a part of ESPRIT Basic Research Action "Logical Frameworks". It has been supported by STU (Swedish National Board for Technical Development).

The *logical* interpretation we have in mind of these simple definitions is given in terms of a *sequent* calculus. A *sequent* is an expression of the form $X \vdash A$ where X is a finite set of conditions. Let X, A be short for $X \cup \{A\}$:

$$X, a \vdash a$$

$$X \vdash \top \qquad X, \bot \vdash C$$

$$A \subset U \left\{ \quad \frac{X, a \vdash C}{X, A \vdash C} \quad (a \in A) \qquad \frac{X \vdash a \quad (a \in A)}{X \vdash A} \right.$$

$$\frac{X \vdash A}{X \vdash a} \quad (A \in D(a)) \qquad \frac{X, A \vdash C \quad (A \in D(a))}{X, a \vdash C}$$

Let $\vdash_D$ be the notion of $D-$consequence defined by the clauses above, i.e. $\vdash_D$ is the smallest relation closed under the given conditions. We will write $X \vdash_D A$ to indicate that $X \vdash A$ holds. Let

$$Def(D) = \{a \mid \top \vdash_D a\},$$

$$\overline{Def(D)} = \{a \mid a \vdash_D \bot\}.$$

Given a definition D and a condition C we define (D, C):

$$\begin{array}{rcll} C & = & \top & \\ A & = & a & (A \in D(a)) \\ a & = & A & (a \in A \subset U) \end{array}$$

Let

$$a \leq_D b \ \text{ if } \ a \in Def((D, b)),$$

$$D \mid a = \{b = B \mid b \leq_D a\}$$

Since $\leq_D$ is a preorder we have $a \leq_D b$ iff $D \mid a \subset D \mid b$. Clearly a is a $\leq_D$-minimal object iff $D(a) = \{\bot\}$ or $D(a) = \{\top\}$. Let

$$\bot(a) = \{b \mid D(b) = \{\bot\} \& b \leq_D a\},$$

$$\top(a) = \{b \mid D(b) = \{\top\} \& b \leq_D a\}.$$

The *base* of a simple definition is its set of $\leq_D$-minimal objects. In what follows we let $D+D'$ denote set union and $D - D'$ set difference. We will also use the notation $D \oplus D'$ to denote the definition $D + \{(a = A) \in D' \mid a \notin Dom(D)\}$. If $X \subset U$ we identify X with the definition $\{x = \top \mid x \in X\}$.

3 Definitions and interpretations

By a *definition* in general we understand a triple $(Dom, Codom, D)$ where

(i) Dom and $Codom$ are two sets such that $Dom \subset Codom$,

(ii) D is a set of equations $a = A$ where $a \in Dom$ and $A \in Codom$. We will also write $Dom(D)$ and $Codom(D)$ sometimes and we often use D to denote the triple $(Dom, Codom, D)$.

A *category* of definitions is then a class $\mathcal{C}$ of definitions together with an operator HOM such that $HOM(D, D')$ is a set of mappings $\phi : Codom \to Codom'$ satisfying

(i) $id \in HOM(D, D)$,

(ii) if $\phi \in HOM(D, D')$ and $\psi \in HOM(D', D'')$, then $\psi\phi \in HOM(D, D'')$,

(iii) if $a \in Dom$, then $\phi(a) \in Dom'$,

(iv) $\phi(D(a)) = D'(\phi(a))$.

Take the natural notion of a category of simple definitions over a given universe U as an example. Then $Dom = U$ for some set U and $Codom = U \cup P(U) \cup \{\perp\top\}$. If D and D' are two simple definitions over U, then $\phi \in HOM(D, D')$ iff

(i) ϕ is a mapping from $Codom$ to $Codom'$ preserving Dom,

(ii) $\phi(\perp) = \perp$,

(iii) $\phi(\top) = \top$,

(iv) $\phi(A) = \{\phi(a) \mid a \in A\}$ for $A \subset U$,

(v) $\phi(D(a)) = D'(\phi(a))$.

By the category of *n-tuples* of simple definitions we understand n-tuples of simple definitions $(D_1, \ldots, D_n)$ over a common universe U. The HOM-set of two such tuples is the intersection of the HOM-sets of the components.

Let $\mathcal{C}$ be a category of definitions. Given a set X we define the category $\mathcal{CX}$:
Objects:

- (D, T) where
  - D is a definition in $\mathcal{C}$,
  - $T : X \to Codom(D)$.

Morphisms:

- (ϕ, f) where
  - $\phi \in HOM(D, D')$,
  - f injective map in $X \to X$ such that

$$\begin{array}{ccc} X & \xrightarrow{f} & X \\ \downarrow{\scriptstyle T} & & \downarrow{\scriptstyle T'} \\ Codom & \xrightarrow{\phi} & Codom' \end{array}$$

$1_{(D,X,T)}$ is of course $(1_D, id)$ and composition is given by the obvious choice.

By a n-ary X-interpretation of $\mathcal{C}$ we mean a functor $\star$ from the category $\mathcal{CX}$ into the category of n-tuples of simple definitions $(D_1, \ldots, D_n)$ which to each object (D,T) associate a map $T^\star : U \to Codom(D)$ such that

$$\begin{array}{ccc} U & \xrightarrow{\phi^\star} & U' \\ \downarrow{\scriptstyle T^\star} & & \downarrow{\scriptstyle T'^\star} \\ Codom & \xrightarrow{\phi} & Codom' \end{array}$$

We will call $p \in U$ *closed* if

$$p \in Def(D_1) \cap \ldots \cap Def(D_n)$$

and say that $p \in U$ is *open* if

$$p \in \overline{Def(D_1)} \cup \ldots \cup \overline{Def(D_n)}$$

An interpretation generates a copy of $Codom(D)$ in terms of the partition of U given by $T^\star$ where

$$C_T = \{p \mid T^\star(p) = C\}.$$

When there is no risk of confusion we simply write $p \in C$. Given a set $V \subset U$ let

$$\begin{array}{rcl} (V,i) & = & \{q \mid \exists p \in V(p \leq_i q\} \\ (i,V) & = & \{q \mid \exists p \in V(q \leq_i p)\} \end{array}$$

Let

$$\begin{array}{rcl} cl(C) & = & \{p \in C \mid p \text{ is closed}\} \\ op(C) & = & \{p \in C \mid p \text{ is open}\} \end{array}$$

We use $v, w, \ldots$ to denote functions in $Codom(D)_T \to Codom(D)_T$ such that $v(C) \subset C$. Each v is a restriction refining the partition given by $T^\star$. We will use the notation $C|v$ for $v(C)$.

We say that v satisfies D in n in the given interpretation if

$$\bigcup_{A \in D(a)} (n, A|v) \subset (n, a|v)$$

and

$$(a|v, n) \subset \bigcup_{A \in D(a)} (A|v, n).$$

Intuitively $((D_1, \ldots, D_n, T^\star), v)$ *models* D in the given interpretation with respect to a certain choice of ordering $\leq_n$ and we write

$$v \models_n D.$$

Given D let

$$\begin{array}{rcl} Def_\star(D) & = & \{a \mid \exists p(p \in a|cl)\} \\ Cov_\star(D) & = & \{C \mid \exists p(p \in C|cl)\}. \end{array}$$

Now if $cl \models D$, then clearly

$$a \in Def_\star(D) \text{ iff } D(a) \cap Cov_\star(D) \neq \emptyset.$$

So a cl-model gives a correct interpretation modulo a notion of a *correct* interpretation of the proper conditions in $Codom(D)$. Now if ϕ is onto, then

$$\phi^\star(C_T) = \phi(C)_{T'}.$$

and

$$\phi^\star(D(a)_T) = D'(\phi^\star(a_T))_{T'}.$$

Assume $p \in \phi^\star(C_T)$, then $p = \phi^\star(q)$ for some q such that $T^\star(q) = C$. Now $\phi(T^\star(q)) = T'^\star(\phi^\star(q))$, so $p \in \phi(C)_{T'}$. Assume $p \in \phi(C)_{T'}$, then $T'^\star(p) = \phi(C)$. Since $\phi^\star$ is onto $p = \phi^\star(q)$ for some q. $T'^\star(\phi^\star(q)) = \phi(C) = \phi(T^\star(q))$, i.e. $q \in C_T$. So we know that $\phi^\star(A_T) = \phi(A)_{T'}$ for $A \in D(a)$. Thus $D'(\phi^\star(a_T))_{T'} = D'(\phi(a)_{T'})_{T'}$ and we are done.

4 Partial inductive definitions

The category of *partial inductive definitions* $\mathcal{P}$ is given as follows: From a set Dom we build $Codom$ using the following formal constructions:

- $\perp$ and $\top$ are objects in $Codom$,
- if $C_i \quad (i \in I)$ is in $Codom$, then $(C_i)_{i\in I}$ is in $Codom$,
- if C and C' are in $Codom$, then $C \to C'$ is in $Codom$. The morphisms that we will consider here has to commute with these formal constructions, i.e.

$$\begin{array}{rcl} \phi(\perp) & = & \perp \\ \phi(\top) & = & \top \\ \phi((C_i)_{i\in I}) & = & (\phi(C_i))_{i\in I} \\ \phi(C \to C') & = & \phi(C) \to \phi(C') \end{array}$$

This notion of isomorphism is strong enough to preserve all basic intensional properties of such a definition. The problem is that it is a too formal notion. It simply treat the basic condition constructions as different *formal* constructions, i.e. the content is rather empty what concerns the intrinsical structure given by various conditions in a definition. So we will use interpretations to try to read off this structure from the structure given by the interpretation. In what follows we will consider a particular binary interpretation of the category of partial inductive definitions $(D_\star, D^\star, T^\star)$. This interpretation will be given as a sort of *completion* of a *flat* Curry-Howard interpretation of these definitions as type systems. Or if you like a completion of a flat natural deduction interpretation of these definitions as systems of introduction rules. This interpretation is *flat* in the sense that abstraction and substitution both are given in terms of local operators. One consequence of this is that although the interpretation is semantical in nature it is given in terms of simple definitions, i.e. natural deduction without global discharge functions. There is a very rich structure hidden in a function calculus given by the standard Curry-Howard interpretation of a definition as a type system, but if we want to make this structure precise there are problems with the usual formulation since things are muddled up in side conditions and various external constructions that has no clear structural meaning. In the kind of interpretation considered here you want of course to express these constructions in a precise manner so to speak *inside* the model itself. This means that we have to introduce a lot of new objects to get a space rich enough for all the distinctions we want to make. The interpretation given below will be carried out in the following steps

(i) we construct a simple definition $K(D,T)$ together with $T^\star$ which so to speak give the *syntax*,

(ii) we then construct another simple definition $L(D,T)$ over $K(D,T)$ which gives the *semantics*,

(iii) $D^\star$ is then $L(D,T) \oplus (K(D,T) - X)$ and $D_\star$ is $K(D,T) - \hat{X}$ where $\hat{X}$ is the formal dual to X.

5 $K(D,T)$

Assume that D and T are given. Let $E(a)$ be the set of equations defining a in D and let $e, e', \ldots$ denote equations. Define

$$e(a) = \begin{cases} A, & \text{if } e \in E(a) \text{ and } e \text{ is } (a = A), \\ 0, & \text{otherwise.} \end{cases}$$

Consider the following definition $KT(D,T)$:

$$\begin{array}{rcl}
\top : \top & = & \top \\
\bot(C,p) : C & = & p : \bot \\
x : T(x) & = & \top \qquad (x \in X) \\
\hat{x} : T(x) & = & \top \qquad (x \in X) \\
F(x,p) : C \to C' & = & \{x : C, p : C'\} \\
\overline{F}(p,q) : C' & = & \{p : C \to C', q : C\} \\
V(p_i)_I : (C_i)_I & = & \{p_i : C_i\}_I \\
\overline{V}(i,p) : C_i & = & p : (C_i)_I \\
D(e,p) : a & = & p : A \qquad (e(a) = A) \\
\overline{D}(p; (x_e, q_e)E(a)) : C & = & \{p : a, q_e : C(e \in E(a)), x_e : e(a)(e \in E(a))\} \\
[x = p]q : C & = & \{x : C', p : C', q : C\}
\end{array}$$

It is clear that if $p : C \in Def(KT(D,T))$, then C is unique. Now let

$$U = \{p \mid \exists\ Cp : C \in Def(KT(D,T))\}.$$

Let

$$\begin{array}{rcl}
F(p : C) & = & p \\
F(A) & = & \{F(a) \mid a \in A\} \\
F(\bot) & = & \bot \\
F(\top) & = & \top
\end{array}$$

We can then define $K(D,T)$ by

$$K(D,T)(F(a)) = \{F(A) \mid A \in KT(D,T)(a)\}$$

A *canonical form* of an object in $K(D,,T)$ is one of the following

$$\top, F(x,p), V(p_i)_I, D(e,p).$$

6 $L(D,T)$

In order to specify $L(D,T)$ we first define a certain subset W of U:

$$\begin{array}{rcl}
\overline{F}(F(x,p),q) & = & \top \\
\overline{V}(i,V(p_i)_I) & = & \top \\
[x=p]q & = & \top \\
\overline{D}(D(e,p);(x_e,q_e)_{E(a)}) & = & \top \\
\bot(C,p) & = & p \\
\overline{F}(p,q) & = & p \\
\overline{V}(i,p) & = & p \\
\overline{D}(p;(x_e,q_e)E(a)) & = & p
\end{array}$$

Now we define a function R on W:

$$\begin{array}{rcl}
R(\overline{F}(F(x,p),q) & = & [x=q]p \\
R(\overline{V}(i,V(p_i)_I)) & = & p_i \\
R(\overline{D}(D(e,p);(x_e,q_e)_{E(a)})) & = & [x_e=p]q_e \\
R(\bot(C,p) & = & \bot(C,R(p)) \\
R(\overline{F}(p,q)) & = & \overline{F}(R(p),q) \\
R(\overline{V}(i,p)) & = & \overline{V}(i,R(p)) \\
R(\overline{D}(p;(x_e,q_e)_{E(a)})) & = & \overline{D}(R(p);(x_e,q_e)_{E(a)}) \\
R([x=p]q) & = & [x=p]R(q) \quad \text{when } q \in W
\end{array}$$

Assume $q \notin W$:

$$R([x=p]z) = \begin{cases} [x=p]p & \text{if } z = x, \\ z, & \text{otherwise.} \end{cases}$$

$$\begin{array}{rcl}
R([x=p]\top) & = & \top \\
R([x=p]\bot(C,q)) & = & \bot(C,[x=p]q) \\
R([x=p]\hat{z}) & = & \hat{z} \\
R([x=p]F(y,r) & = & F(y,r') \\
 & & \text{where } r' = [x=p]r \text{ if } x \neq y,\ r \text{ otherwise} \\
R([x=p]\overline{F}(r,s)) & = & \overline{F}([x=p]r,[x=p]s) \\
R([x=p]V(p_i)_I) & = & V([x=p]p_i)_I \\
R([x=p]\overline{V}(i,r)) & = & \overline{V}(i,[x=p]r) \\
R([x=p]D(e,r)) & = & D(e,[x=p]r) \\
R([x=p]\overline{D}(r;(x_e,s_e)_{E(a)})) & = & \overline{D}([x=p]r;(x_e,s'_e)_{E(a)} \\
 & & \text{where } s' = [x=p]s \text{ if } x \neq x_e,\ s \text{ otherwise}
\end{array}$$

$L(D, T)$ is then the following simple definition

$$\begin{array}{rcl} F(x,p) & = & [x = \hat{x}]p \\ \overline{D}(p; (x_e, q_e)_{E(a)}) & = & \{p, [x = \hat{x}]q_e \quad (e \in E(a))\} \\ & & \text{where } \overline{D}(p; (x_e, q_e)_{E(a)}) \notin W \\ p & = & R(p) \quad \text{for } p \in W \end{array}$$

7 The interpretation $\star$

$D_\star$ and $D^\star$ are given as

$$\begin{array}{rcl} D_\star & = & K(D,T) - \hat{X} \\ D^\star & = & L(D,T) \oplus (K(D,T) - X) \end{array}$$

and $T^\star$ is defined in the obvious way by

$$T^\star(p) = \text{the unique } C \text{ such that } p : C \in Def(KT(D,T)).$$

So assume $\phi \in HOM(D, D')$, then we define $\phi^\star(p)$ and show $\phi(T^\star(p)) = T'^\star(\phi^\star(p))$ by recursion and induction on $KT(D, T)$:

$$\phi^\star(x) = f(x)$$

we know that $\phi(T(x)) = T'(f(x))$, but $\phi^\star(x) = f(x)$ and $T^\star(x) = T(x)$.

$$\phi^\star(\hat{x}) = \widehat{f(x)}$$

$$\phi^\star(\top) = \top$$

$$\phi^\star(\perp(C, p)) = \perp(C\phi^\star(p))$$

$$\phi^\star(F(x,p)) = F(f(x), \phi^\star(p))$$

$$\begin{array}{rcl} \phi(T^\star(p)) & = & \phi(T^\star(x) \to T^\star(q)) \\ & = & \phi(T^\star(x)) \to \phi(T^\star(q)) \end{array}$$

We have

$$\begin{array}{rcl} T'^\star(\phi^\star(p)) & = & T'^\star(F(f(x), \phi^\star(q))) \\ & = & T'^\star(f(x)) \to T'^\star(\phi^\star(q)) \end{array}$$

and by IH

$$\begin{array}{rcl} \phi(T^\star(x)) & = & T'^\star(\phi^\star(x)) \\ \phi(T^\star(q)) & = & T'^\star(\phi^\star(q)) \end{array}$$

the cases $\overline{F}, V, \overline{V}$ can be treated in a similar fashion.

$$\phi^\star(\overline{F}(p,q)) = \overline{F}(\phi^\star(p), \phi^\star(q))$$

this make sense since by IH $\phi(T^\star(p)) = \phi^\star(T'^\star(p))$ and $\phi(T^\star(q)) = \phi^\star(T'^\star(q))$.

$$\begin{array}{rcl} \phi^\star(V(p_i)_I) & = & V(\phi^\star(p_i))_I \\ \phi^\star(\overline{V}(i,p)) & = & \overline{V}(i,\phi^\star(p)) \end{array}$$

that this make sense follows directly from IH.

$$\begin{array}{rcl} \phi^\star(D(e,p)) & = & D'(\phi(e),\phi^\star(p)) \\ \phi^\star(\overline{D}(p;(x_e,q_e)_{E(a)}) & = & \overline{D'}(\phi^\star(p);(f(x)_{\phi(e)},\phi^\star(q)_{\phi(e)})_{E(\phi(a))}) \\ \phi^\star([x=p]q) & = & [f(x)=\phi^\star(p)]\phi^\star(q) \end{array}$$

it follows from IH and the fact that $\phi \in HOM(D,D')$ that this make sense. Now $\phi(T^\star(p)) = \phi(a)$ where e is $(a = T^\star(q))$. We have $T'^\star(\phi^\star(p)) = T'^\star(D(\phi(e),\phi^\star(q))$ and $\phi(e)$ is $(\phi(a) = \phi(T^\star(q)))$, so the result follows directly from IH. The $\overline{D}$-case and the case when $p = [x = q]r$ also follows directly in a similar fashion from IH.

We extend $\phi^\star$ by

$$\begin{array}{rcl} \phi^\star(\bot) & = & \bot \\ \phi^\star(\top) & = & \top \\ \phi^\star(A) & = & \{\phi^\star(a) \mid a \in A\} \end{array}$$

It easy to see that $\star$ is a functor from $\mathcal{PX}$ to the category of pairs of simple definitions. Obviously $\phi^\star$ is in $HOM(D_\star, D'_\star)$. We observe that $p \in W(D)$ iff $\phi^\star(p) \in W(D')$. This is a simple induction on W and then $\phi^\star(D^\star(p)) = D'^\star(\phi^\star(p))$ follows by direct inspection of the various cases. So $\phi^\star \in HOM(D^\star, D'^\star)$. If $\phi = 1_{(D,T)}$, then clearly $\phi^\star = 1_{D^\star}$. It is also clear that $(\psi\phi)^\star = \psi^\star\phi^\star$. So $\star$ gives an interpretation of the partial inductive definitions.

Let us say that an object p is *convergent* with respect to a set $V \subset U$ if $\forall q \leq_n p \exists r \in V(r \leq_n q)$. A function v will be called *proper* if

(i) all objects in $C|v$ are convergent with respect to $\overline{W}$,

(ii) there are no open objects in $C|v$,

(iii) $q \in A|v$ iff $D(e,q) \in a|v$,

(iv) $p \in a|v$ implies $R(p) \in a|v$.

Canonical examples of proper functions are cl and

$$conv(C) = \{p \in C \mid p \text{ is convergent w.r.t. } \overline{W} \text{ and not open}\}.$$

If v is a proper function, then $v \models D$ in $\star$. Let v be a proper function and assume $p \in (2, A|v)$ we have $p \leq^\star q$ for some $q \in A|v$, then clearly $D(e,q) \in a$ where e is $a = A$ and obviously $q \leq^\star D(e,q)$. So $p \leq^\star D(e,q)$ and $D(e,q) \in a|v$ which follows from the closure properties of v Assume $p \in (a|v, 2)$, i.e. there is a $q \in a|v$ such that $q \leq^\star p$. It is easy to see that if $q \notin W$, then either q is an object in canonical form or an open object. If q is in canonical form, then $q = D(e,r)$ for some $e \in E(a)$ and $r \in A$ for some $A \in D(a)$. Now clearly $r \leq^\star p$ and $r \in (A|v, 2)$ since $q \in a|v$. q is by definition not open, so assume $q \in W$. q is convergent with respect to $\overline{W}$. So $R^n(q) \in \overline{W}$ for some n. Since q is not open this means that $s = R^n(q)$ is in canonical form and since R "preserves types" this in turn means that $s = D(e,t)$ for some s and t. We have $t \leq^\star p$ and $t \in A|v$ for some $A \in D(a)$ which follows from the closure properties of v.

It is easy to see that if ϕ is onto, then $\phi^\star$ is also onto and if ϕ is an isomorphism, then $\phi^\star$ is also an isomorphism.

The standard Curry-Howard interpretation can be obtained from $KT(D,T)$ by deleting clauses for $\hat{x}$ and $[x=p]q$. Let us call this definition $KT^-(D,T)$. We will define a translation j from $Def(D^\star)$ to $Def(KT^-(D,T)$ and show that $j(p) : T^\star(p) \in Def(KT^-(D,T))$:

$$\begin{array}{rcl} j(\hat{x}) & = & x \\ j(\top) & = & \top \\ j(\bot(C,p)) & = & \begin{cases} j(R(\bot(C,p))), & \text{if } \bot(C,p) \in W, \\ \bot(C,j(p)), & \text{otherwise.} \end{cases} \end{array}$$

We have $j(\hat{x}) = x : T(x) = T(\hat{x})$.

$$\begin{array}{rcl} j(F(x,p) & = & F(x, j([x=\hat{x}]p)) \\ j(\overline{F}(p,q)) & = & \begin{cases} j(R(\overline{(}p,q)), & \text{if } \overline{F}(p,q) \in W, \\ \overline{F}(j(p),j(q)), & \text{otherwise.} \end{cases} \end{array}$$

$T^\star(F(x,p)) = T(x) \to T^\star(p)$. $j(F(x,p)) = F(x, j([x=\hat{x}]p))$. Now by IH $j([x=\hat{x}]p) : T^\star([x=\hat{x}]p)$. Since $T^\star([x=\hat{x}]p) = T^\star(p)$ we have $j(F(x,p)) : T(x) \to T^\star(p)$.
$F(p,q)$. Assume $F(p,q \in W$. By IH $j(R(\overline{F}(p,q)) : T^\star(R(F(p,q)$. Since R "preserves types" it follows directly that $j(F(p,q) : T^\star(F(p,q))$. If $F(p,q) \notin W$, then $j(F(p,q)) = \overline{F}(j(p),j(q))$. By IH $j(p) : T^\star(p)$Jand $j(q) : T^\star(q)$, so clearly $j(F(p,q)) : T^\star(F(p,q))$. The other cases are treated in a similar manner. So the definition make sense as a translation to $KT^-(D,T)$:

$$\begin{array}{rcl} j(V(p_i)_I) & = & V(j(p_i))_I \\ j(\overline{(}k,p)) & = & \begin{cases} j(R(\overline{(}k,p))), & \text{if } \overline{V}(k,p) \in W, \\ \overline{V}(k,j(p)), & \text{otherwise.} \end{cases} \\ j(D(e,p)) & = & D(e,j(p)) \\ j(\overline{D}(p;(x_e,q_e)_{E(a)})) & = & \begin{cases} j(R(\overline{D}(p;(x_e,q_e)_{E(a)})), & \text{if } \overline{D}(p;(x_e,q_e)_{E(a)}) \in W, \\ \overline{D}(j(p);(x_e, j([x=\hat{x}]q_e)_{E(a)}), & \text{otherwise.} \end{cases} \\ j([x=p]q) & = & j(R([x=p]q)) \end{array}$$

If p is closed, then $j(p)$ will be a *closed term*. Let $Var(t)$ denote the usual notion of *free variables* in a term t. Reasoning by induction on $D^\star$ we see that $Var(j(p)) \subset j(\bot_{D_\star}(p))$:
The case when $p = \widehat{f(x)}, \top$ is trivial. The case $\bot(C,q)$ follows directly from IH. So assume $p = F(x,q)$. We have

$$\begin{array}{rcl} Var(j(p)) & = & Var(j([x=\hat{x}]q)) - \{x\} \\ Var(j(p)) & \subset & Var(j([x=\hat{x}]q)) \subset_{IH} j(\bot([x=\hat{x}]q)) \end{array}$$

and

$$\bot(p) \supset \bot([x=\hat{x}]q) - \{x\}.$$

$p = \overline{F}(q,r)$. Assume $p \in W$. Now in general if $p \in W$, then $\bot(R(p)) \subset \bot(p)$. So

$$Var(j(p)) = Var(j(R(p)) \subset_{IH} j(\bot(R(p)) \subset j(\bot(p)).$$

So this is the general pattern when $p \in W$.
If $p \notin W$ the result follows from IH using

$$\begin{aligned} Var(j(p)) &= Var(j(q)) \cup Var(j(r)) \\ \perp(p) &= \perp(q) \cup \perp(r). \end{aligned}$$

If $p = V(p_i)_I$, then we use the fact that

$$\begin{aligned} Var(j(p)) &= \bigcup_I Var(j(p_i)) \\ \perp(p) &= \bigcup_I \perp(p_i) \end{aligned}$$

We treat $p = \overline{V}(i, q)$ similarly to $\overline{F}(q, r)$.
$p = D(e, q)$. We have

$$Var(j(p)) = Var(j(q)) \subset_{IH} j(\perp(q)) = j(\perp(p)).$$

$p = \overline{D}(q; (x_e, r_e)_{E(a)})$. Assume $p \notin W$. Then

$$\begin{aligned} Var(j(p)) &= Var(j(q)) \cup (\bigcup_{E(a)} Var(j(r_e)) - \{x_e\} \\ \perp(p) &= \perp(q) \cup (\bigcup_{E(a)}(\perp(r_e)) \end{aligned}$$

and by IH

$$\begin{aligned} Var(j(q)) &\subset j(\perp(q)) \\ Var(j([x_e = \hat{x_e}]r_e)) &\subset j(\perp([x_e = \hat{x_e}]r_e)) \end{aligned}$$

References

[1] P. Aczel. *An introduction to inductive definitions.* In: Handbook of Mathematical Logic, ed. J. Barwise, North Holland, Amsterdam, 1977.

[2] D. Fredholm and S. Serafimovski. *Partial inductive definitions as type systems for lambda terms.* In Proceedings of the 1989 Båstad workshop in programming logic, Department of Computer Science, Chalmers 1990.

[3] L. Hallnäs. *Partial inductive definitions.* To appear in TCS.

[4] L. Hallnäs. *On the syntax of infinite objects: an extension of Martin-Löfs theory of expressions.* In Proceedings of COLOG-88, eds. P. Martin-Löf, G. Mints, Springer Lecture Notes in Computer Science 1990.

[5] J-Y. Girard. *Towards a geometry of interaction.* In Proceedings of AMS conference on categories, logic and computer science , Boulder 1987.

[6] P. Martin-Löf. *Notes on constructive mathematics.* Almqvist & Wiksell, Stockholm 1970.

[7] D. Prawitz. *Towards a foundation of a general proof theory.* In: Logic, Methodology and the Philosophy fo Sciences IV, ed. P. Suppes, North Holland, Amsterdam, 1973.

[8] D. Prawitz. *On the idea of a general proof theory.* Synthese 27, 1974.

Goal Directed Proof Construction in Type Theory

LEEN HELMINK and RENÉ AHN
Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, the Netherlands

Abstract. In this paper, a method is presented for proof construction in Generalised Type Systems. An interactive system that implements the method has been developed.

Key words. Type Theory, Generalised Type Systems, Constructive Type Theory, Calculus of Constructions, Typed Lambda Calculus, Natural Deduction.

1 Introduction

Generalised type systems (GTSs) [Ba89] provide a uniform way to describe and classify type theoretical systems. A method is presented to perform unification based top down proof construction for generalised type systems, thus offering a well-founded, elegant and powerful underlying formalism for a proof development system. It combines the advantages of Horn clause resolution and higher order natural deduction style theorem proving. No theoretical contribution to generalised type systems is claimed. First, the method will be explained for constructive type theory, also known as the Calculus of Constructions [Co85]. Then, the method will be generalised to generalised type systems. The method thus applies to many differents variants of type theory, e.g. systems in the families of AUTOMATH [Br73][Da80], LF [Ha87], Elf [Pf89]. A full derivation example is included. A proof environment based on the method, named *Constructor*, has been developed within Esprit project 1222: 'Genesis' [He89][He90]. Experiments with this system demonstrate the power and efficiency of the method.

The method presented offers sound inference steps for goal directed proof construction in Generalised Type Systems. Because proof construction is not decidable, strategic information has to be provided by users, either in the form of interactive choices, or in the form of algorithms (so-called *tactics*). In a method that is unification based, this strategic information is reduced. This allows simple user interaction and tactics programming for proof systems based on the method. Although the provided inference steps suggest certain tactics and user interaction modes, these issues are outside the scope of this paper.

A type theory presents a set of rules to derive types of objects in a given context with assumptions about the type of primitive objects. The objects and types are expressions in typed λ-calculus. The *propositions as types* paradigm provides a direct mapping between (higher order) logic and type theory. In this interpretation, contexts correspond to theories, types correspond to propositions, and objects correspond to proofs of propositions. Type theory has successfully demonstrated its capabilities to formalize many parts of mathematics in a uniform and natural way. Moreover, the type structure is rich enough to serve as a meta language to formalize object logics. For many variants of type theory, types are strongly normalizing and types of objects are unique. Together with the derivability of types of objects, this guarantees decidability of the typing relation. Note that this permits automatic proof checking in higher order logic. Several proof checkers have been developed for this purpose, and have already been used on extensive examples [Ju76],[Da80],[Co85].

The problem addressed in this paper is to construct an object in a given context, given its type. This amounts to higher order theorem proving. This paper demonstrates that this construction problem can be handled by Horn clause resolution, provided that the set of available Horn clauses is continuously adapted to the context in which the proof is conducted. This rests on a mechanism that provides a simple clausal interpretation for the assumptions in a context.

2 Constructive Type Theory (CTT)

We assume familiarity with the (typed) λ-calculus [Ba80]. Constructive type theory is a variant of the *propositions as types* paradigm, which is based on the fact that there exists an elegant analogy between simply-typed λ-calculus and minimal propositional logic. This section gives a short overview of the basic principles and describes the particular generalised type system of interest. We shall use a system developed by Coquand, a version of the Calculus of Constructions. In GTS terminology, this system is referred to as $\lambda C'$ [Ba89]. More about such systems can be found in [Ba89], [Br73], [Cq85], [Fo83], [Hu87]. In the subsequent sections, the method will be explained for this system. Then, a generalisation of the method will be given for Generalised Type Systems.

2.1 Terms

The syntactic formation rules for the terms in the system are defined as:

- constant, viz. one of $\{prop,\ type,\ kind\}$.
- variable, denoted by an identifier.
- $\lambda[x{:}A].B$, typed abstraction, where A and B are terms, and x an identifier denoting a variable.
- $\Pi[x{:}A].B$, generalized Cartesian product of types B indexed over x of type A, where A and B are terms, and x an identifier denoting a variable. [1]
- $(A\ B)$, application, where A and B are terms (function and argument). [2]

[1] In type theory, typed abstraction is often denoted $[x{:}A]B$, while typed product is denoted $(x{:}A)B$ or $\{x{:}A\}B$.
[2] Application associates to the left, so we will write (a b c) for ((a b) c).

The terms in the system are typed, and types are terms themselves. This imposes a hierarchy of types. Four levels of expressions are distinguished in the type system: 0-, 1-, 2-, and 3-expressions [3], where n-expressions serve as types of $(n+1)$-expressions $(n = 0, 1, 2)$. A *typing* is a construction of the form $[t{:}T]$, where t and T are terms. The intuition behind this is that T is the type of t. There is only one 0-expression: the predefined constant 'supertype' *kind*. This is regarded as the zeroth level of the type hierarchy [4]. We introduce two predefined constants as primitive 1-expressions (kinds) of the system: The kind *type*, the set containing all 'plain' types, and the kind *prop*, that plays an identical role and is treated similarly, but which is inhabited by propositions (assertions) [5]. Because *kind* is the only 0-expression, we have $[type{:}kind]$ and $[prop{:}kind]$.

Later, when the precise typing rules for terms are given, it turns out that the terms at the different levels satisfy the following general structure:

$$
\begin{array}{lll}
\text{0-expressions: } O & ::= & kind \\
\text{1-expressions: } K & ::= & type \mid prop \mid \Pi[x{:}U].K \\
\text{2-expressions: } A & ::= & id \mid \lambda[x{:}U].A \mid \Pi[x{:}U].A \mid (AM) \\
\text{3-expressions: } V & ::= & id \mid \lambda[x{:}U].V \mid (VM)
\end{array}
$$

where the metavariable O ranges over 0-expressions, K ranges over 1-expressions, A over 2-expressions, U over 1- and 2-expressions, M ranges over 2- and 3-expressions, V over 3-expressions, x and id range over variables.

2.2 Correctness

Correctness of typings containing free variables is always relative to a certain *context*, consisting of a (possibly empty) list of assumptions, in which the free variables have to be explicitly introduced with their type. Thus, a context is of the form:

$$[x_1{:}U_1],\ldots,[x_n{:}U_n] \quad (\text{n}\geq 0)$$

No variable may be declared more than once in a context. Moreover, the free variables occurring in U_i must have been declared earlier in the context, i.e. $\mathcal{FV}(U_i) \subseteq \{x_1, \ldots, x_{i-1}\}$, where $\mathcal{FV}(U)$ stands for the set of variables occurring free in U. So contexts are constructed in a certain order, and adding an assumption introduces a new scope. Apart from assumptions, contexts may contain

[3] For 1-, 2-, and 3-expressions the classification *kinds*, *types* and *values* is also in vogue, but as these names are overloaded the AUTOMATH terminology is adopted here to avoid confusion.

[4] This level is an auxiliary one, introduced to ensure typings for all terms of interest.

[5] In many versions of this system, *type* and *prop* are identified (usually denoted '$\star$' or '*Type*'). Here we will explicitly distinguish between them, to avoid confusion in the 'propositions as types' interpretation. This is however not essential to the method.

definitions. A definition introduces a variable as an abbreviation for an arbitrary correct expression. This variable can be referred to thereafter. A definition is of the form $[c \equiv a{:}A]$, where c is a fresh identifier, and establishes that c abbreviates the term a of type A.

We will use Γ as a metavariable over contexts. Well-formedness of a context Γ will be denoted *well-formed*(Γ). We will write $\Gamma, [x{:}U]$ to denote the context Γ extended with the assumption $[x{:}U]$, and Γ_1, Γ_2 for the concatenation of contexts Γ_1 and Γ_2. The symbol '$\emptyset$' represents the empty context. We will write $E \in \Gamma$ to denote that E is a member of context Γ. A *sequent* is an expression of the form $\Gamma \vdash [a{:}A]$, denoting that $[a{:}A]$ is a correct typing in the well-formed context Γ. The predicate *constant*(x) denotes that x is one of the predefined constants *type*, *prop* or *kind*. We will write $B[a/x]$ to denote substitution of the term a for the free occurrences of the variable x in the expression B. We will write '$=_{\beta\delta}$' to denote the transitive reflexive closure of β- and δ-reduction. β-reduction corresponds to the usual notion in typed lambda-calculus, and δ-reduction denotes local expansion of definitions (unfolding). Note that both β- and δ-reduction are context-dependent. We ignore here all problems concerning renaming of bound variables (in case of clash of variable names): all variable names are considered unique here, and all equality is modulo α-conversion. This can be achieved by using De Bruijn indices [Br72] or Barendregt's variable convention [Ba80].

Well-formedness of contexts and correct typing of terms is inductively defined as:

[0] *well-formed*$(\emptyset)$

[1a] If *well-formed*(Γ) then $\Gamma \vdash [\mathit{type}{:}\mathit{kind}]$

[1b] If *well-formed*(Γ) then $\Gamma \vdash [\mathit{prop}{:}\mathit{kind}]$

[2a] If $\Gamma \vdash [A{:}K]$ and *constant*(K) then *well-formed*$(\Gamma, [x{:}A])$

[2b] If $\Gamma \vdash [a{:}A]$ then *well-formed*$(\Gamma, [c \equiv a{:}A])$

[3a] If *well-formed*(Γ) and $[x{:}A] \in \Gamma$, then $\Gamma \vdash [x{:}A]$

[3b] If *well-formed*(Γ) and $[c \equiv a{:}A] \in \Gamma$, then $\Gamma \vdash [c{:}A]$

[4] If $\Gamma, [x{:}A] \vdash [b{:}B]$ then $\Gamma \vdash [\lambda[x{:}A].b : \Pi[x{:}A].B]$

[5] If $\Gamma, [x{:}A] \vdash [B{:}K]$ and *constant*(K) then $\Gamma \vdash [\Pi[x{:}A].B{:}K]$

[6] If $\Gamma \vdash [a{:}A]$ and $\Gamma \vdash [b : \Pi[x{:}A].B]$ then $\Gamma \vdash [(b\ a){:}B[a/x]]$

[7] If $\Gamma \vdash [a{:}A]$ and $\Gamma \vdash [B{:}K]$ and *constant*(K) and $A =_{\beta\delta} B$ then $\Gamma \vdash [a{:}B]$

Explanation

Rule 0 *(empty)* says that the empty context is well-formed.

Rule 1 *(kinds)* says that the primitives *type* and *prop* are correct kinds in any well-formed context.

Rule 2 *(introduction)* says that if A is a domain type (explained below), then we can introduce a new variable of type A in the context. The variable may not be bound already in the context, but α problems are superficial and ignored here. Domains or domain types are precisely those expressions, that have type *type*, *prop* or *kind*. Note that domains are all 1-expressions and all 2-expressions except for the ones that normalize to lambda abstractions. The intuition behind this is that domains are inhabitable types, i.e. those expressions that can serve as the type of 2- and 3-expressions. Although lambda abstractions over 2-expressions are 2-expressions themselves, they can not play the role of inhabited types, i.e. there are no 3-expressions that have a lambda abstracted type. For this reason, abstraction over terms only occurs with variables that are in a domain type. Note that *kind* is not a domain (it has been introduced as an auxiliary term that allows typings for 1-expressions). Further, rule 2b says that for correct typings, a definition may be added to the context.

Rule 3 *(selection)* says that assumptions from the context are correct in the context. Moreover, definition names have the type of the object they abbreviate.

Rule 4 *(lambda abstraction)* says that given a correct type or value, we can construct a term with a product type by doing lambda abstraction over it with a (correctly) typed variable.

Rule 5 *(pi abstraction)* says that given a correct type, we can do pi abstraction over it with a correctly typed variable and obtain a term with the same type.

Rule 6 *(application)* says that we can do application with a function and another term, that has the same type as the domain of the function.

Rule 7 *(type conversion)* says that types and kinds remain correct after conversion under the transitive reflexive closure of β- and δ-reduction.

2.3 Properties

The described system has the following properties [6]:

It has the Church-Rosser property under beta and delta reduction. Moreover, strong normalization holds. This implies that every reduction sequence always terminates and every term has a unique normal form. This property guarantees decidability of correctness for sequents.

Unicity of types:

[6] Although these properties have been proved for a system not containing definitions, the inclusion of definitions does not affect any of them, as they can always be removed by expansion.

If $\Gamma \vdash [a{:}A]$ and $\Gamma \vdash [a{:}B]$ then $A =_{\beta\delta} B$.

The system is monotonic, i.e. derivable typings remain derivable under context extensions.

The system also satisfies the important property of *substitutivity*:

If $\Gamma_1, [x{:}A], \Gamma_2 \vdash E$ and $\Gamma_1 \vdash [a{:}A]$ then $\Gamma_1, \Gamma_2[a/x] \vdash E[a/x]$.

Another property is 'closure under reduction', i.e. correctness invariance of sequents under *beta*-reductions of objects (this is sometimes referred to as subject reduction). Hence:

If $\Gamma \vdash [a{:}A]$ and $a \succ_\beta b$ then $\Gamma \vdash [b{:}A]$

where $\succ_\beta$ stands for beta reduction.

The system is closed under outermost η-reduction on objects:

If $\Gamma \vdash [\lambda[y{:}B].\,(C\ y) : D]$, where y does not occur free in C, then $\Gamma \vdash [C{:}D]$.

A proof of this property is given in appendix A. Though this property may seem far-fetched, it is essential for the soundness of our proof construction method.

For details and examples the reader is referred to the references [Ba89], [Co85], [Cq85], [Hu87], [Ju76].

2.4 Interpretation and Use

If, for a dependent product type $\Pi[x{:}A].B$, x does not occur free in B ($x \notin \mathcal{FV}(B)$), the type simplifies, as usual, to the ordinary *function type* $A{\rightarrow}B$, the type of functions that map objects of type A to objects of type B. Types are considered sets of objects, and '$[x{:}A]$' (x has type A) is interpreted as membership of x of the set A. In case $A{:}prop$, A is considered a proposition and a construction $a{:}A$ is interpreted as: a is a proof for A, i.e. a proposition plays the role of the type of its proofs. This means that a proposition is considered valid if and only if it is inhabited. Thus, proving a proposition A comes down to constructing an a such that $a{:}A$. By associating in this way objects with proofs, and types with propositions, it turns out that the system described contains predicate logic. More precisely, it contains intuitionistic higher order predicate logic. Below, this is elucidated by presenting the rules for universal quantification ($\forall$), implication ($\Rightarrow$) and negation.

If B:*prop*, then $\Pi[x{:}A].B$ can be interpreted as the universally quantified proposition $\forall x{:}A.B$. If x does not occur free in B and A:*prop* and B:*prop*, then $\Pi[x{:}A].B$ can be interpreted as the intuitionistic implication $A \Rightarrow B$. Implication associates to the right, so $(A \Rightarrow B \Rightarrow C)$ stands for $(A \Rightarrow (B \Rightarrow C))$.

Now the correctness rules for this system entail the introduction and elimination rules for implication and universal quantification in natural deduction style:

- implication introduction:
  If under an assumption $x{:}A(:prop)$, and possibly other assumptions not depending on x, an object $b{:}B(:prop)$ can be derived ($x \notin \mathcal{FV}(B)$), then $x{:}A$ can be *discharged* (removed from the context) and $\lambda[x{:}A].b$ proves $A \Rightarrow B$.

- implication elimination (modus ponens):
  Under assumptions $p{:}(A \Rightarrow B)$ and $x{:}A(:prop)$, we can conclude $(px){:}B$.

- universal quantifier introduction:
  If under an assumption (context element) $x{:}A$, and possibly other assumptions not depending on x, an object $b{:}B(:prop)$ can be derived, then $x{:}A$ can be *discharged* and $\lambda x{:}A.b$ proves $\forall x{:}A.B$.

- universal quantifier elimination (specialization):
  Under assumption $p{:}\forall x{:}A.B$, if $a{:}A$ we can conclude $(pa){:}B[a/x]$.

The natural deduction system for predicate logic that is obtained for free in this system, also contains negation. First, contradiction is identified as the absurd assertion that all propositions hold [7]:

$$[falsum \equiv \Pi[a{:}prop].\ a : prop]$$

i.e. *falsum* is by definition $\forall a : prop.\ a$. The contradiction rule becomes:

$$[contradiction \equiv \lambda[a{:}prop].\ \lambda[f{:}falsum].\ (f\ a) : \Pi[a{:}prop].\ \Pi[f{:}falsum].\ a]$$

i.e. *contradiction* abbreviates a proof of $\forall a : prop.\ (falsum \Rightarrow a)$. Negation of a proposition can now be expressed with the function:

[7] The system is consistent in the sense that, in the empty context, no proof can be derived for this proposition.

$$[not \equiv \lambda[a{:}prop].\ \Pi[p{:}a].\ falsum : \Pi[a{:}prop].\ prop]$$

i.e. $(not\ a)$ denotes $(a \Rightarrow falsum)$. In order to obtain full classical logic in our intuitionistic framework, we can introduce the classical double negation law by adding an assumption to the context:

$$[dnl : \Pi[a{:}prop].\ \Pi[p{:}(not\ (not\ a))].\ a]$$

i.e. *dnl* axiomatically introduces $\forall a : prop.\ ((not\ (not\ a)) \Rightarrow a)$.

The Calculus of Constructions formalism thus provides a definition language that can be and has been used to formalize and mechanically verify many parts of mathematics. Texts in this language are written in the form of *theories* (*books* in AUTOMATH terminology). Theories are contexts as introduced in the previous section. For a field of interest, assumptions allow the axiomization of the primitive notions, whereas the definitions allow abbreviation of derived notions like lemmas. A theory is correct if it obeys the given correctness rules for a well-formed context.

2.5 Example

As an example, consider the following theory Γ_0:

$$
\begin{array}{l}
[nat : type], \\
[0 : nat], \\
[s : \Pi[x{:}nat].\ nat], \\
[< : \Pi[x{:}nat].\ \Pi[y{:}nat].\ prop] \\
[axiom1 : \Pi[x{:}nat].\ (<\ x\ (s\ x))], \\
[trans : \Pi[x{:}nat].\ \Pi[y{:}nat].\ \Pi[z{:}nat]. \\
\qquad\quad \Pi[p{:}(<\ x\ y)].\ \Pi[q{:}(<\ y\ z)].\ (<\ x\ z)], \\
[ind : \Pi\ [p : \Pi[x{:}nat].\ prop]. \\
\qquad \Pi\ [g : (p\ 0)]. \\
\qquad \Pi\ [h : \Pi[n{:}nat].\ \Pi[hyp{:}(p\ n)].\ (p\ (s\ n))]. \\
\qquad \Pi\ [z : nat].\ (p\ z)], \\
[pred1 \equiv \lambda[x{:}nat].\ (<\ 0\ (s\ x)) : \Pi[x{:}nat].\ prop], \\
[baseproof \equiv (axiom1\ 0) : (pred1\ 0)], \\
[lemma1 \equiv \lambda[x{:}nat].\ \lambda[y{:}nat].\ \lambda[p{:}(<\ x\ y)].\ (trans\ x\ y\ (s\ y)\ p\ (axiom1\ y)) : \\
\qquad\qquad \Pi[x{:}nat].\ \Pi[y{:}nat].\ \Pi[p{:}(<\ x\ y)].\ (<\ x\ (s\ y))], \\
[stepproof \equiv \lambda[n{:}nat].\ \lambda[p{:}(pred1\ n)].\ (lemma1\ 0\ (s\ n)\ p) : \\
\qquad\qquad \Pi[n{:}nat].\ \Pi[p{:}(pred1\ n)].\ (pred1\ (s\ n))], \\
[result \equiv (ind\ \ pred1\ \ baseproof\ \ stepproof) : \Pi[x{:}nat].\ (pred1\ x)]
\end{array}
$$

Interpretation:

nat is a *type*.
0 is a *nat*.
s is a function from *nat* to *nat* (the successor function).
$<$ is a binary predicate over *nat*s.
axiom1 states that $\forall x : nat.\ (< x\ (s\ x))$.
trans states transitivity for $<$: $\forall x, y, z : nat.\ (< x\ y) \Rightarrow (< y\ z) \Rightarrow (< x\ z)$.
ind introduces the induction axiom for natural numbers: for all predicates p over *nat*s, given a proof of $(p\ 0)$ and a proof of $\forall x : nat.\ (p\ n) \Rightarrow (p\ (s\ n))$, we can conclude $\forall x : nat.\ (p\ x)$.

These are the primitive notions of this theory. The rest of the theory is aimed at proving the theorem $\forall x : nat\ (< 0\ (s\ x))$.

pred1 is the associated predicate over *nat*s: $\lambda x : nat.\ (< 0\ (s\ x))$.
baseproof abbreviates a proof that this predicate holds for the number *0*.
lemma1 proves $\forall x, y : nat.\ (< x\ y) \Rightarrow (< x\ (s\ y))$.
stepproof proves $\forall n : nat.\ ((p\ n) \Rightarrow (p\ (s\ n)))$ for the predicate in question.
result finally proves $\forall x : nat\ (< 0\ (s\ x))$ by application of induction to the predicate, the *baseproof* and the *stepproof*.

From now on, the reader is invited to make the interpretations for himself.

3 Resolution Inference

In goal directed proving, the main idea is to start off with the goal to be proven, and to replace this goal by appropriate subgoals by application of inference rules. Horn clause inference rules consist of a set of antecedents $A_1 \ldots A_k$ (referred to as the premises or conditions of the rule) and one conclusion A (the head of the rule). The antecedents A_i and the conclusion A are predicates over terms from the universe of discourse. In this paper, Horn clauses will be denoted:

$$A \Leftarrow A_1\ ...\ A_k$$

A Horn clause is interpreted as: (A_1 and ... and A_k) *implies* A (where $k \geq 0$). A is a logical consequent of the premises in the rule. A Horn clause is *valid* if it holds in the universe of discourse. For $k = 0$, A is unconditionally valid, i.e. a 'fact'. Terms in Horn clauses may contain logical variables, that are considered to be universally quantified over the clause. Logical variables play the role of placeholders for terms during proof construction.

A derivation tree is a term tree constructed with Horn clauses. It derives the top term of the tree. Horn clauses are the building blocks for derivation trees, they play the role of tree combinators.

Unification of two terms t_1 and t_2 is the process of finding a substitution σ of terms for the logical variables in both terms, such that $[t_1]_\sigma = [t_2]_\sigma$, where '=' represents an equality relation of interest. σ is called a unifier for t_1 and t_2. $[t]_\sigma$ is the term obtained from t by simultaneously replacing the variables in t by the associated terms given in σ. $[t]_\sigma$ is called an instance of t.

Unification determines how two Horn clauses can be combined inside a derivation tree. A derivation tree for a term is either a Horn clause concluding that term, or is a derivation tree where an antecedent term is replaced by a derivation tree for that antecedent term, after unifying both derivation trees. If all the leaves of a derivation tree are facts, the derivation is complete and serves as a justification for the root term.

A derivation tree can be *collapsed* by removing all nodes except for the root and the leaves: the tree is replaced with the root term having the leaves as subnodes. Collapsed derivation trees, called derivations for short, have the same interpretation as Horn clauses. They correspond to derived Horn clauses.

Let $\mathcal{R}_1$ be the derivation:

$$\mathcal{R}_1 \ : \ A \Leftarrow A_1 \ ... \ A_n$$

and $\mathcal{R}_2$ be the Horn clause:

$$\mathcal{R}_2 \ : \ B \Leftarrow B_1 \ ... \ B_m$$

Let θ be a unifier for A_i and B, i.e. $[A_i]_\theta = [B]_\theta$. Resolution [Ro65] now states that the derivation

$$\mathcal{R}_{1,2} \ : \ [B \Leftarrow A_1 \ ... \ A_{i-1} \ B_1 \ ... \ B_m \ A_{i+1} \ ... \ A_n]_\theta$$

is a valid consequent *(resolvent)* of $\mathcal{R}_1$ and $\mathcal{R}_2$.

4 CTT Proof Construction Method

It is a well-known fact, that correctness of sequents $\Gamma \vdash [t : T]$ (t has type T in context Γ) in Constructive Type Theory and related systems is decidable, even feasably decidable, and several proof checkers exist that mechanically determine correctness (well-formedness) for given CTT theories [Ju76], [Da80], [Co85]. For a proof construction system, the objective is not to verify whether a given object has a certain given type, but, for a given type, to attempt construction of an inhabitant of this type. For propositions, this corresponds to finding a proof object.

The starting point for goal directed or top down construction of typings is backward chaining with inference rules, where proof obligations for goals are replaced by proof obligations for appropriate new subgoals.

The central problem with goal directed proof construction in Constructive Type Theory is that direct backward chaining with the correctness rules of section 2.2 is hardly possible. Therefore, the approach is to extract from the given formalization a sound set of derived rules, that do allow easy backward inference. These derived rules then serve as the primitive proof steps of the system.

There exists a direct translation from intuitionistic theories to Horn clauses. This is the central idea of the method presented and it allows resolution based proof construction. The proof objects constructed by the method are fully compatible with the input expected (and approved of) by CTT checkers. This implies that we remain faithful to constructive type theory and its usual features and semantics, without introduction of any ad hoc extensions, thus keeping the underlying formalism as clean as possible.

In this section, first the basic principles of resolution based inference will be summarized. Then, the core of the proof construction method for CTT will be described, by formalizing all correct basic derivation steps.

In the method, CTT sequents will be derived using Horn clause resolution. The problem of interest is to find an object of a given type. More precisely: given a context (theory) Γ and a type A, the objective is to construct an object p such that $\Gamma \vdash [p{:}A]$.

To this end, we will first need a notion of logical variables in addition to the regular CTT terms. Logical variables play the role of meta variables over CTT terms. To avoid confusion with CTT variables, we will denote logical variables by identifiers prefixed will a '$\sharp$' symbol. A term is grounded if it does not contain logical variables. A context will be called grounded if it contains grounded terms only.

Then, the queries considered consist of one goal of the form $\Gamma \vdash [p{:}A]$. For such a goal, the following well-formedness properties are required: Γ must be a grounded, well-formed context, A must be a grounded, correct domain type, and p is either a logical variable or a grounded CTT term. In the latter case, the problem reduces to type checking. This is the same problem as verifying whether a definition $[c \equiv p : A]$ is a correct extension to context Γ.

For a given query Q, the derivation process starts with the trivial derivation '$Q \Leftarrow Q$', which is obviously valid. The objective is to transform this derivation by resolution with valid Horn clauses, until the derivation '$Q' \Leftarrow$' is reached. Q' is then a correct instance of Q. For a query $\Gamma \vdash [p{:}A]$ this implies that an object of the requested type has been constructed.

Both derivations and Horn clauses will be of the form:

$$\Gamma \vdash [p{:}A] \Leftarrow \Gamma_1 \vdash [p_1{:}A_1] \ \ldots \ \Gamma_n \vdash [p_n{:}A_n].$$

Derivations and Horn clauses may contain logical variables that play the role of meta variables over CTT terms. They are not to be confused with CTT variables. The meaning of a derivation or a clause is that *if* instantiations for the logical variables can be found, such that the antecedent sequents are correct, *then* the associated consequent is a correct sequent.

The following invariant properties will hold for all derivations (but not necessarily for Horn clauses):

1. all contexts Γ and $\Gamma_1 \ldots \Gamma_n$ will be grounded and well-formed.
2. $\Gamma \subseteq \Gamma_1$, ... , $\Gamma \subseteq \Gamma_n$.
3. for any logical variable $\sharp P$ occurring in the type field A_i of a subgoal $\Gamma_i \vdash [p_i{:}A_i]$, $\sharp P \in \{p_1 \ldots p_{i-1}\}$. If an object field p_i of a subgoal $\Gamma_i \vdash [p_i{:}A_i]$ is not a logical variable, then for any logical variable $\sharp P$ occurring in p_i, $\sharp P \in \{p_1 \ldots p_{i-1}\}$.
4. for any logical variable $\sharp P$ occurring in the object field p of the conclusion $\Gamma \vdash [p{:}A]$, $\sharp P \in \{p_1 \ldots p_n\}$.

The first property reflects the fact that construction always takes place within a known context. The second property states that contexts can be extended during backward proving. The third property ensures that logical variables are 'introduced before use'. The fourth property guarantees that the conclusion of a derivation will be grounded when all subgoals have been solved (the type field A of a derivation conclusion is grounded from the start). For our queries $\Gamma \vdash [\sharp P{:}A]$ this implies that an object $\sharp P$ of the requested type A has been constructed. Note that the trivial derivations that correspond to our queries of interest have all the required properties.

It turns out that we can avoid a problem that usually arises with inference rules over sequents, viz. unification over contexts. The reason for this is that in contrast to general purpose higher order theorem provers as presented in e.g. [Pa86], [Pa89] and [Fe88], the method inferences at the object level, not at the meta level. This is possible because the method is specialized for type theory only; it is not a generic inference method over arbitrary sequents. Contexts will be treated in a special way, and it is sufficient to unify over typings. Our goals, that denote CTT sequents, will be regarded as tuples with a typing and a context.

For the method to work, it is necessary that contexts occurring in derivations are grounded, so that (1) we do not unify over contexts at all, and (2) we can extract the necessary object level horn clauses directly from contexts. The exact consequences of this restriction will be alluded later.

In the subsequent sections, valid Horn clauses will be derived. These Horn clauses will not violate the given invariant property for derivations. Some of the Horn clauses correspond directly to correctness rules for type theory and are of interest to all subgoals in a derivation. The problem is that no suitable Horn clause can be given for the application rule (rule 6), on account of the substitution. A solution for this problem is presented, that rests on a mechanism to provide a direct clausal interpretation for the assumptions in a context. The Horn clauses thus obtained cover derivation steps that can construct the necessary application terms. This is the quintessence of the method. For any given subgoal $\Gamma_i \vdash [p_i{:}A_i]$, this mechanism, when applied to the context Γ_i, allows derivation of a set of valid Horn clauses that are candidates for resolution on this particular subgoal. An early proposal for this mechanism can be found in [Ah85].

4.1 Unification and Type Conversion

Unification determines whether a clause is applicable in a given situation. Unification will also handle the type conversion rule (rule 7), dealing with equality of types. It is important to observe that it is sufficient to provide unification over typings, not over contexts (although context information is of course relevant for β- and δ-reductions during unification). Unifying typings $[P{:}A]$ and $[P'{:}A']$ will be achieved by unifying the objects P and P' and subsequently unifying the types A and A'. For types, the unification is with respect to β- and δ-equality. Although β-equality for objects is not explicit in the correctness rules, it is also desirable to identify β- (and δ-) equivalent terms. This is justified by the closure under reduction property, and corresponds to proof normalization [Co85][Da80][Ha87]. Because we unfold derived clauses completely, it is desirable to augment object unification with outermost η-equality, to ensure reachability of objects in η-normal form.

Unification for expressions in typed λ-calculus with respect to α, β and possibly η-conversion requires complete higher order unification. For simply typed λ-calculus this problem has a possibly infinite set of solutions and is known to be semidecidable, in the sense that if two terms do not unify, search algorithms for unifiers may diverge [Hu75]. [Hu75] also gives a complete algorithm for this problem in simply typed λ-calculus. In the more complicated case of the Calculus of Constructions, where types are also terms in typed λ-calculus, it is not yet completely clear. Elliott [El89] and Pym [Py88] have independently extended Huets algorithm to dependent types for the logical framework LF (see also [Pf89]). Huet tells us that Dowek [Do90] is working towards extending the algorithm up the Barendregt 'cube' [Ba89], a theory that gives a classification of type systems and that includes the Calculus of Constructions. It is too early to report results. See also section 7.3. For implementations of the method, sound approximations for higher order unification can always be used. Such approximations can be very usable in practice (section 8 gives an example) although they affects completeness, of course.

4.2 Kinds rule

The *kinds* rule is directly equivalent to the valid Horn clauses:

$$\Gamma \vdash [type : kind] \Leftarrow .$$
$$\Gamma \vdash [prop : kind] \Leftarrow .$$

For any subgoal $\Gamma_i \vdash [p_i{:}A_i]$, such a rule applies if the typing $[p_i{:}A_i]$ unifies with $[type{:}kind]$ or $[prop{:}kind]$. Γ is not treated as a logical variable. Because contexts in our derivations are always grounded and well-formed, the well-formedness check on Γ_i (required in rule 1) is needless.

4.3 Lambda abstraction rule

The lambda abstraction rule (rule 4) corresponds to the valid Horn clause:

$$\Gamma \vdash [\lambda[x{:}\sharp A].\sharp B : \Pi[x{:}\sharp A].\sharp T] \Leftarrow \Gamma, [x{:}\sharp A] \vdash [\sharp B{:}\sharp T].$$

where $\sharp A, \sharp B, \sharp T$ are logical variables. Remember that Γ is not treated as a logical variable. The typing of a goal of the form $\Gamma_i \vdash [P : \Pi[x{:}A].T]$ may be unified with the typing in the conclusion of this rule, unifying P with $\lambda[x{:}\sharp A].\sharp B$ and resulting in the new stripped subgoal $[\sharp B{:}\sharp T]$, to be solved in the context Γ_i extended with the typing $[x{:}\sharp A]$. Thus, application of this rule introduces a local context extension for goals. To ensure that this new context is grounded and well-formed, the restriction is imposed that A must be grounded and correct domain, thus preventing logical variables over domains to be introduced in the context. For example, this rule can not be used to find a proof for an implication $\sharp A \Rightarrow B$, because this would introduce an unknown assumption $[p{:}\sharp A]$ in the context.

4.4 Pi abstraction rule

The pi abstraction rule (rule 5) corresponds to the valid Horn clause:

$$\Gamma \vdash [\Pi[x{:}\sharp A].\sharp B : \sharp K] \Leftarrow \Gamma, [x{:}\sharp A] \vdash [\sharp B{:}\sharp K].$$

where $\sharp A, \sharp B, \sharp K$ are logical variables. Again, Γ does not play the role of a logical variable. For goals of the form $\Gamma_i \vdash [\Pi[x{:}A].B : K]$, application of this rule results, after unification of typings, in a stripped subgoal $[B{:}K]$, to be solved in the context Γ_i extended with the typing $[x{:}A]$. K must be a constant. Though this check has to be postponed if K is not grounded, this can be circumvented by providing three different pi abstraction rules for $K = kind$, *type* and *prop*, respectively. To ensure that the new context is grounded and well-formed, again the restriction is imposed that A must be a grounded and correct domain. The restriction prevents e.g. using this rule on a goal $\Gamma_i \vdash [\sharp P{:}prop]$.

4.5 Derived Clauses

The application rule (rule 6) cannot be translated directly to a Horn clause, on account of the substitution. A solution is offered, for which the following theorem is essential:

Main Theorem

Correctness of a sequent of the form

$$\Gamma \vdash [C : \Pi[x_1{:}A_1] \; ... \; \Pi[x_n{:}A_n].B]$$

is equivalent to the validity of the Horn clause:

$$\begin{aligned}\Gamma, \Gamma' \vdash [(C \; \sharp x_1 \; ... \; \sharp x_n) : B[\sharp x_1/x_1, ..., \sharp x_n/x_n]] \Leftarrow\; & \Gamma, \Gamma' \vdash [\sharp x_1{:}A_1] \\ & \Gamma, \Gamma' \vdash [\sharp x_2{:}A_2[\sharp x_1/x_1]] \\ & \cdots \\ & \Gamma, \Gamma' \vdash [\sharp x_n{:}A_n[\sharp x_1/x_1, ..., \sharp x_{n-1}/x_{n-1}]].\end{aligned}$$

where $\sharp x_1 \; ... \; \sharp x_n$ are the logical variables of the Horn clause. The context Γ, Γ' denotes any well-formed context extension of Γ. Note that all possible occurrences of the CTT variables x_i have been replaced by corresponding logical variables $\sharp x_i$. A complete proof of this theorem is given in appendix A. Intuitively, the theorem is a version of the heuristic application principle.

For $n = 0$, the set of premises is empty and the application in the consequent simplifies to the object C.

The selection rule (rule 3a and 3b) justifies that the theorem is in particular applicable to all introductions and definitions occurring in any well-formed context Γ, i.e.

For all CTT variables c, if

$$[c : \Pi[x_1{:}A_1] \; ... \; \Pi[x_n{:}A_n].B] \in \Gamma \quad \text{or} \quad [c \equiv C : \Pi[x_1{:}A_1] \; ... \; \Pi[x_n{:}A_n].B] \in \Gamma$$

this theorem guarantees that:

$$\begin{aligned}\Gamma \vdash [(c \; \sharp x_1 \; ... \; \sharp x_n) : B[\sharp x_1/x_1, ..., \sharp x_n/x_n]] \Leftarrow\; & \Gamma \vdash [\sharp x_1{:}A_1] \\ & \Gamma \vdash [\sharp x_2{:}A_2[\sharp x_1/x_1]] \\ & \cdots \\ & \Gamma \vdash [\sharp x_n{:}A_n[\sharp x_1/x_1, ..., \sharp x_{n-1}/x_{n-1}]].\end{aligned}$$

This result is now used to interpret the introductions and definitions in a context in clausal form, thus providing the possible application candidates. The intuition behind this is to 'unfold' the top level pi abstractions for context elements to clauses. For any goal $\Gamma \vdash E$, all clauses thus obtained from the context Γ are available as valid Horn clauses for resolution on this goal. Note that the associated contexts unify directly.

Although for a given context element of type $\Pi[x_1{:}A_1]$ ... $\Pi[x_n{:}A_n].B$ (where B is not itself a Π-abstraction) this gives $n+1$ different valid Horn clauses, it is sufficient to provide a completely unfolded clause with n antecedents. If the clause has been unfolded too far to unify directly with a sequent, this can be compensated by first resolving the sequent with the lambda abstraction rule.

Note that this mechanism now covers both the application rule and the selection rule.

Example

The context Γ_0 from the example in section 2.6 gives rise to the following valid Horn clauses:

$$
\begin{array}{l}
[\mathit{nat} : \mathit{type}] \Leftarrow . \\
[\mathit{0} : \mathit{nat}] \Leftarrow . \\
[(s\ \sharp X) : \mathit{nat}] \Leftarrow [\sharp X : \mathit{nat}]. \\
[(<\ \sharp X\ \sharp Y) : \mathit{prop}] \Leftarrow [\sharp X : \mathit{nat}]\ [\sharp Y : \mathit{nat}]. \\
[(\mathit{axiom1}\ \sharp X) : (<\ \sharp X\ (s\ \sharp X))] \Leftarrow [\sharp X : \mathit{nat}]. \\
[(\mathit{trans}\ \sharp X\ \sharp Y\ \sharp Z\ \sharp P\ \sharp Q) : (<\ \sharp X\ \sharp Z)] \Leftarrow [\sharp X{:}\mathit{nat}]\ [\sharp Y{:}\mathit{nat}]\ [\sharp Z{:}\mathit{nat}] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\sharp P{:}(<\ \sharp X\ \sharp Y)]\ [\sharp Q{:}(<\ \sharp Y\ \sharp Z)]. \\
[(\mathit{ind}\ \sharp P\ \sharp G\ \sharp H\ \sharp Z) : (\sharp P\ \sharp Z)] \Leftarrow [\sharp P : \Pi[x{:}\mathit{nat}].\,\mathit{prop}]\ [\sharp G : (\sharp P\ \mathit{0})] \\
\qquad\qquad\qquad\qquad\qquad\qquad [\sharp H : \Pi[n{:}\mathit{nat}].\,\Pi[\mathit{hyp}{:}(\sharp P\ n)].\,(\sharp P\ (s\ n))]\ [\sharp Z : \mathit{nat}]. \\
[(\mathit{pred1}\ \sharp X) : \mathit{prop}] \Leftarrow [\sharp X : \mathit{nat}].
\end{array}
$$

where the context parameter Γ_0 has been omitted in the sequents, because it is identical for typings in the consequent and in the antecedents of a clause and therefore it will not affect the contexts of goals upon resolution.

The next section will demonstrate the method by giving a top down derivation of the example proof presented in section 2.5. Appendix B gives a number of practical extensions to the method.

5 Derivation Example

To elucidate the method, a top down derivation is presented for the theorem that is proved in example 2.6. To this end, we start off with a context Γ_0 containing the axioms given there:

$$
\begin{array}{l}
[\mathit{nat} : \mathit{type}], \\
[\mathit{0} : \mathit{nat}], \\
[s : \Pi[x{:}\mathit{nat}].\ \mathit{nat}], \\
[< \,: \Pi[x{:}\mathit{nat}].\ \Pi[y{:}\mathit{nat}].\ \mathit{prop}], \\
[\mathit{axiom1} : \Pi[x{:}\mathit{nat}].\ (<\ x\ (s\ x))], \\
[\mathit{trans} : \Pi[x{:}\mathit{nat}].\ \Pi[y{:}\mathit{nat}].\ \Pi[z{:}\mathit{nat}]. \\
\qquad\quad \Pi[p{:}(<\ x\ y)].\ \Pi[q{:}(<\ y\ z)].\ (<\ x\ z)], \\
[\mathit{ind} : \Pi\ [p : \Pi[x{:}\mathit{nat}].\ \mathit{prop}]. \\
\qquad \Pi\ [g : (p\ \mathit{0})]. \\
\qquad \Pi\ [h : \Pi[n{:}\mathit{nat}].\ \Pi[\mathit{hyp}{:}(p\ n)].\ (p\ (s\ n))]. \\
\qquad \Pi\ [z : \mathit{nat}].\ (p\ z)], \\
[\mathit{pred1} \equiv \lambda[x{:}\mathit{nat}].\ (<\ \mathit{0}\ (s\ x)) : \Pi[x{:}\mathit{nat}].\ \mathit{prop}]
\end{array}
$$

The goal to prove is: $\forall x{:}\mathit{nat}.(\mathit{pred1}\ y)$, i.e. a proof object $\sharp P$ is sought, such that:

$\Gamma_0 \vdash [\sharp P : \Pi[y{:}\mathit{nat}].\ (\mathit{pred1}\ y)]$.

The bottom up proof in example 2.5 uses lemmas. In proof systems based on the method, extension of the context with derived lemmas is of course be permitted, but here we will construct the proof directly.

The associated trivial derivation for the query is:

$$\Gamma_0 \vdash [\sharp P : \Pi[y{:}\mathit{nat}].\ (\mathit{pred1}\ y)] \Leftarrow \Gamma_0 \vdash [\sharp P : \Pi[y{:}\mathit{nat}].\ (\mathit{pred1}\ y)].$$

The Horn clauses available for resolution are:

$$
\begin{array}{l}
\Gamma \vdash [\lambda[x{:}\sharp A].\sharp B : \Pi[x{:}\sharp A].\sharp T] \Leftarrow \Gamma, [x{:}\sharp A] \vdash [\sharp B{:}\sharp T]. \\
\Gamma \vdash [\Pi[x{:}\sharp A].\sharp B : \sharp K] \Leftarrow \Gamma, [x{:}\sharp A] \vdash [\sharp B{:}\sharp K].
\end{array}
$$

viz. the lambda abstraction rule and the pi abstraction rule (the kinds rule is of no relevance to this example). These Horn clauses are always available for goals. For a given goal, they are extended with Horn clauses that can be obtained from the unfolded context elements of the goal. For the antecedent in the given trivial derivation this means:

$$
\begin{array}{l}
[\mathit{nat} : \mathit{type}] \Leftarrow . \\
[\mathit{0} : \mathit{nat}] \Leftarrow . \\
[(s\ \sharp X) : \mathit{nat}] \Leftarrow [\sharp X : \mathit{nat}]. \\
[(<\ \sharp X\ \sharp Y) : \mathit{prop}] \Leftarrow [\sharp X : \mathit{nat}]\ [\sharp Y : \mathit{nat}]. \\
[(\mathit{axiom1}\ \sharp X) : (<\ \sharp X\ (s\ \sharp X))] \Leftarrow [\sharp X : \mathit{nat}]. \\
[(\mathit{trans}\ \sharp X\ \sharp Y\ \sharp Z\ \sharp P\ \sharp Q) : (<\ \sharp X\ \sharp Z)] \Leftarrow [\sharp X{:}\mathit{nat}]\ [\sharp Y{:}\mathit{nat}]\ [\sharp Z{:}\mathit{nat}] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\sharp P{:}(<\ \sharp X\ \sharp Y)]\ [\sharp Q{:}(<\ \sharp Y\ \sharp Z)]. \\
[(\mathit{ind}\ \sharp P\ \sharp G\ \sharp H\ \sharp Z) : (\sharp P\ \sharp Z)] \Leftarrow [\sharp P : \Pi[x{:}\mathit{nat}].\ \mathit{prop}]\ [\sharp G : (\sharp P\ \mathit{0})] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\sharp H : \Pi[n{:}\mathit{nat}].\ \Pi[\mathit{hyp}{:}(\sharp P\ n)].\ (\sharp P\ (s\ n))]\ [\sharp Z : \mathit{nat}]. \\
[(\mathit{pred1}\ \sharp X) : \mathit{prop}] \Leftarrow [\sharp X : \mathit{nat}].
\end{array}
$$

where Γ_0 has been omitted from the sequents. Similar clauses can be constructed for extensions of Γ_0. The only rule applicable to our derivation is the lambda abstraction rule. Resolution gives:

$$\Gamma_0 \vdash [\sharp P : \Pi[y{:}nat].\,(\mathit{pred1}\ y)] \Leftarrow \Gamma_0, [y{:}nat] \vdash [\sharp P' : (\mathit{pred1}\ y)].$$

instantiating $\sharp P$ to $\lambda[y{:}nat].\sharp P'$. In the extended context of the subgoal, a derived clause for y ($[y{:}nat] \Leftarrow .$) is now available. Resolution with the induction clause (*ind*) from the context gives:

$$\begin{aligned}\Gamma_0 \vdash [\sharp P : \Pi[y{:}nat].(\mathit{pred1}\ y)] \Leftarrow\ & \Gamma_0, [y{:}nat] \vdash [\mathit{pred1} : \Pi[x{:}nat].\,prop]\\ & \Gamma_0, [y{:}nat] \vdash [\sharp G : (\mathit{pred1}\ \mathit{0})]\\ & \Gamma_0, [y{:}nat] \vdash [\sharp H : \Pi[n{:}nat].\,\Pi[hyp{:}(\mathit{pred1}\ n)].\,(\mathit{pred1}\ (s\ n))]\\ & \Gamma_0, [y{:}nat] \vdash [y : nat].\end{aligned}$$

instantiating $\sharp P'$ to $(\mathit{ind}\ \sharp P''\ \sharp G\ \sharp H\ \sharp Z)$, $\sharp P''$ to *pred1*, and $\sharp Z$ to y. Note that this requires higher order unification. The first subgoal is grounded and can be checked, but this goal can also be solved after resolution with the lambda abstraction rule, provided that the unification knows that *pred1* is equivalent to $[x{:}nat](\mathit{pred1}\ x)$ (outermost η-equivalence). The last subgoal is solved directly with the Horn clause for y from the context. The derivation thus becomes:

$$\begin{aligned}\Gamma_0 \vdash [\sharp P : \Pi[y{:}nat].(\mathit{pred1}\ y)] \Leftarrow\ & \Gamma_0, [y{:}nat] \vdash [\sharp G : (\mathit{pred1}\ \mathit{0})]\\ & \Gamma_0, [y{:}nat] \vdash [\sharp H : \Pi[n{:}nat].\,\Pi[hyp{:}(\mathit{pred1}\ n)].\,(\mathit{pred1}\ (s\ n))].\end{aligned}$$

The first subgoal resolves with the Horn clause for *axiom1* from the context, instantiating $\sharp G$ to $(\mathit{axiom1}\ \mathit{0})$ and leaving $[\mathit{0}{:}nat]$ as trivial subgoal that can be resolved immediately. The remaining subgoal is stripped twice with the lambda abstraction rule. The derivation is now:

$$\Gamma_0 \vdash [\sharp P : \Pi[y{:}nat].\,(\mathit{pred1}\ y)] \Leftarrow \Gamma_1 \vdash [\sharp H' : (\mathit{pred1}\ (s\ n))].$$

instantiating $\sharp H$ to $\lambda[n{:}nat].\lambda[hyp{:}(\mathit{pred1}\ n)].\ \sharp H'$. Γ_1 stands for $\Gamma_0, [y{:}nat], [n{:}nat], [hyp{:}(\mathit{pred1}\ n)]$. The remaining proof obligation is now resolved with the context clause for transitivity (*trans*):

$$\begin{aligned}\Gamma_0 \vdash [\sharp P : \Pi[y{:}nat].\,(\mathit{pred1}\ y)] \Leftarrow\ & \Gamma_1 \vdash [\mathit{0} : nat]\\ & \Gamma_1 \vdash [\sharp Y : nat]\\ & \Gamma_1 \vdash [(s\ (s\ n)) : nat]\\ & \Gamma_1 \vdash [\sharp \mathit{P1} : (<\ \mathit{0}\ \sharp Y)]\\ & \Gamma_1 \vdash [\sharp Q : (<\ \sharp Y\ (s\ (s\ n)))].\end{aligned}$$

instantiating $\sharp H'$ to $(\mathit{trans}\ \mathit{0}\ \sharp Y\ (s\ (s\ n))\ \sharp \mathit{P1}\ \sharp Q)$. The first and third subgoal are eliminated with context clauses from Γ_1 for $\mathit{0}$, s and n. Because these subgoals are grounded, this amounts to checking. Resolving the last subgoal with *axiom1* instantiates $\sharp Q$ to $(\mathit{axiom1}\ n)$ and $\sharp Y$ to $(s\ n)$. The derivation has become:

$$\begin{aligned}\Gamma_0 \vdash [\sharp P : \Pi[y{:}nat].\,(pred1\ y)] \Leftarrow\ & \Gamma_1 \vdash [(s\ n) : nat]\\ & \Gamma_1 \vdash [\sharp P1 : (<\ 0\ (s\ n))]\\ & \Gamma_1 \vdash [(s\ n) : nat].\end{aligned}$$

The proof is completed with the context clauses for s, n and hyp. The complete proof $\sharp P$ is now:

$$\begin{aligned}\lambda[y{:}nat].\,(ind\ pred1\ & \\ & (axiom1\ 0)\\ & \lambda[n{:}nat].\,\lambda[hyp{:}(pred1\ n)].\,(trans\ 0\ (s\ n)\ (s\ (s\ n))\ hyp\ (axiom1\ (s\ n)))\\ & y)\end{aligned}$$

Note that this proof is an η-redex. This is due to the fact that derived clauses are unfolded as far as possible here, thus constructing applications that are provided with the full number of arguments. If outermost η-conversion of objects is provided, the η-normal proof can also be derived.

6 Completeness

An interesting question is whether the method is complete in the sense that a top down derivation can be constructed for all correct inhabitants (modulo object conversion) of a given type (checking is complete). Note that completeness is of course determined by the completeness of the higher-order unification procedure. But what exactly are the consequences of the restriction that we impose on the context, viz. that it is always grounded?

We already saw that our queries of interest are not affected by the restriction. Now consider the effect of the restriction during the inferencing process. It should be clear that only the lambda rule and the pi rule are affected by the restriction, as they may extend a context during resolution. In a sense, the lambda and pi rule are the only rules that do inferencing at the meta level. For all issues related to completeness (at least in a non-deterministic sense), the following observation is important: due to the third invariant property on derivations, we only need to consider goals where the type field of the conclusion is grounded, because any logical variable $\sharp P$ occurring there can be instantiated by first solving the associated goal where $\sharp P$ is introduced. This implies that the partiality of the lambda-abstraction rule poses no fundamental restrictions, because it can be circumvented by postponing resolution on the goal in question. It is clear that the restriction on the applicability of the pi-abstraction rule poses real limitations: it explicitly restricts querying for arbitrary 2-expressions, i.e. it refuses to enumerate all Π-abstracted propositions or types. The expressive power of CTT is such, that it does allow the construction of proofs that involve e.g. induction loading, where a stronger proposition is sought to construct a proof for a weaker one. Top down construction of such proofs is unattainable in general. The limitations imposed on the pi abstraction rule are related to this fundamental problem.

Thus, the method as presented is not complete. The restriction affects construction of propositions and types. Since it is possible to enumerate all propositions (or types), the method can be made complete by replacing the pi rule by an enumerator algorithm, in those places where construction is desired.

7 The Method for Generalised Type Systems

Barendregt has introduced the notion of generalised type systems, including the so-called 'λ-cube' [Ba89][Ba90][Ba9-]. This approach describes many constructive type systems. The classification of systems is obtained by controlling which abstractions are permitted. This section will first summarize the rules for GTS, and then explain how the method is generalised to GTS.

7.1 GTS rules

The syntactic formation rules for the terms in a GTS are defined as :

- constant, viz. c $\in$ C.
- variable, denoted by an identifier.
- $\lambda[x{:}A].B$, typed abstraction.
- $\Pi[x{:}A].B$, typed product.
- $(A\ B)$, application.

A specification of a GTS consists of a triple (S,A,R) such that

- S $\subseteq$ C, called *sorts*.
- A is a set of axioms of the form [c:s], where $c \in C$ and $s \in S$.
- R is a set of rules of the form (s_1, s_2, s_3), with $s_1, s_2, s_3 \in S$.
  A pair (s_1, s_2) abbreviates the rule (s_1, s_2, s_2).

Given a specification (S,A,R), the correctness rules for the associated GTS are defined as:

[0] *well-formed*$(\emptyset)$

[1] If *well-formed*(Γ) then $\Gamma \vdash [c{:}s]$ $\qquad ([c{:}s]\in$A)

[2a] If $\Gamma \vdash [A{:}s]$ then *well-formed*$(\Gamma, [x{:}A])$ $\qquad (s\in$S)

[2b] If $\Gamma \vdash [a{:}A]$ then *well-formed*$(\Gamma, [c \equiv a{:}A])$

[3a] If *well-formed*(Γ) and $[x{:}A] \in \Gamma$, then $\Gamma \vdash [x{:}A]$

[3b] If *well-formed*(Γ) and $[c \equiv a{:}A] \in \Gamma$, then $\Gamma \vdash [c{:}A]$

[4] If $\Gamma \vdash [A{:}s_1]$ and $\Gamma, [x{:}A] \vdash [B{:}s_2]$ and $\Gamma, [x{:}A] \vdash [b{:}B]$
then $\Gamma \vdash [\lambda[x{:}A].b : \Pi[x{:}A].B]$ $\qquad ((s_1, s_2, s_3) \in$ R)

[5] If $\Gamma \vdash [A{:}s_1]$ and $\Gamma, [x{:}A] \vdash [B{:}s_2]$ then $\Gamma \vdash [\Pi[x{:}A].B{:}s_3]$ $\qquad ((s_1, s_2, s_3) \in$ R)

[6] If $\Gamma \vdash [a{:}A]$ and $\Gamma \vdash [b : \Pi[x{:}A].B]$ then $\Gamma \vdash [(b\ a){:}B[a/x]]$

[7] If $\Gamma \vdash [a{:}A]$ and $\Gamma \vdash [B{:}s]$ and $A =_{\beta\delta} B$ then $\Gamma \vdash [a{:}B]$ $\qquad (s\in$S)

Note that the theory of constructions can be described by the following GTS specification:

S	*type*, *prop*, *kind*
A	*type*:*kind*, *prop*:*kind*
R	$\mathbf{S}^2$, i.e. all pairs (s_1, s_2)

This particular system is also referred to as $\lambda C'$. For details and properties of generalised type systems the reader is referred to [Ba89][Ba90].

Extending the method towards Generalised type systems is straightforward. The subsequent sections generalise the valid Horn clauses for CTT from chapter 4 to valid Horn clauses for a given GTS.

7.2 Properties

The type theoretical properties that are essential to the method, viz. monotonicity, substitutivity, and closure under reduction (cf. section 2.3) hold for all generalised type systems. Not all properties that hold for CTT are guaranteed to hold for all generalised type systems. An interesting property that is lost in some generalised type systems, is the property of decidability of proof checking:

- The unicity of types theorem does not hold for all systems.
- In an inconsistent GTS (e.g. the systems known as $\lambda *$ and λU, that contain an inconsistency known as Girard's paradox [Ba9-]), terms exists that have no normal form. The property of strong normalisation is lost.

For such systems, grounded goals should not be handled by a dedicated checking procedure, but be treated as ordinary proof goals. Note that this does not pose problems on the side conditions of the lambda and pi rules, because the correctness check on the introduced domain A has been transferred to a subgoal.

The method extensions for CTT proposed in appendix B are also valid for generalised type systems.

7.3 Unification and Type Conversion

The unification that is required for the method is directly dependent on the GTS at hand. For a discussion on unification within generalised type systems in the λ-cube, the reader is referred to section 4.1. The previous section explained that the properties of unicity of types and strong normalisation do not hold for all generalised type systems. Because equality of grounded terms is not even decidable the unification can never be complete for such systems. It should be clear that for non-normalising systems nothing can be guaranteed, and one will always have to be satisfied with incomplete unification algorithms.

7.4 Axioms

The axioms A from a GTS specification are directly available as inference rules (cf. the kinds rule in section 4.2). The valid Horn clauses are:

$$\Gamma \vdash [c : s] \Leftarrow .$$

One for each associated axiom $[c{:}s]{\in}\mathrm{A}$.

7.5 Lambda abstraction rules

The lambda abstraction rule (rule 4) corresponds to a set of Horn clauses, one for each (s_1, s_2) pair for which there exists an s_3 such that $(s_1, s_2, s_3) \in \mathrm{R}$. The valid Horn clauses are:

$$\begin{aligned}\Gamma \vdash [\lambda[x{:}\sharp A].\sharp B : \Pi[x{:}\sharp A].\sharp T] \Leftarrow\ & \Gamma \vdash [\sharp A{:}s_1]\\ & \Gamma, [x{:}\sharp A] \vdash [\sharp T{:}s_2]\\ & \Gamma, [x{:}\sharp A] \vdash [\sharp B{:}\sharp T].\end{aligned}$$

As in section 4.3, the restriction is imposed that A must be grounded after unification. Note that these new lambda rules contain explicit subgoals to control the allowed abstractions. These now also handle the correctness obligation on A.

7.6 Pi abstraction rules

The pi abstraction rules (rule 5) corresponds to a set of valid Horn clauses:

$$\begin{aligned}\Gamma \vdash [\Pi[x{:}\sharp A].\sharp B : s_3] \Leftarrow\ & \Gamma \vdash [\sharp A{:}s_1]\\ & \Gamma, [x{:}\sharp A] \vdash [\sharp B{:}s_2].\end{aligned}$$

one for each triple $(s_1, s_2, s_3) \in \mathrm{R}$. As in section 4.4, the restriction is imposed that A must be grounded after unification. Note again that these new rules contain explicit subgoals to control the permitted abstractions. These now also handle the correctness obligation on A.

7.7 Derived Clauses

The derived clauses, the main theorem and the proof of the main theorem in Appendix A (that constitute the essence of the method) are not affected by the generalisation, and are valid without modifications for generalised type systems (cf. section 4.5 and Appendix A).

7.8 Completeness

The method can never be complete for all generalised type systems, because no complete unification procedure exists. For systems that do have the properties of unicity of types and strong normalisation, the reader is referred to the discussion in section 6.

8 The *Constructor* Proof Environment

This section gives a short description of an interactive proof environment, named *Constructor*, that implements an inference machine based on the described method. The machine enforces correctness of proof construction in generalised type systems. The mouse-based interface has been built using the *Genesis* system, the tool generator that resulted from Esprit project 1222. Details of the *Constructor* system can be found in [He88][He89][He90]. Here, we will explain its most important features.

When using *Constructor*, there is always a global context present, which is the theory that formalizes a domain of interest. A global context is prefixed with a GTS specification, to identify the type system of interest. The specific GTS inference rules are generated from this specification. The application rule (rule 6) and the selection rule (rule 3), which are handled by the method proposed, are identical for all GTS variants. A proof editor is provided in which conjunctions of queries can be posed, and that admits application of correct proof steps. Queries are interpreted in the global context. Queries are typings of the form $[A{:}B]$. Figures 1 and 2 present some screen images of the system (for the syntax used in the figures see section 8.1).

Both interactive user-guided inference and automatic search are possible and may intermingle. In interactive mode the user may, for a selected goal, choose a clause from a menu with resolution candidates. Optionally, the system checks instantiated subgoals that may arise after resolution steps. Currently only one default search strategy or *tactic* (*tactical* in LCF terminology) is present for automatic search. It uses a consecutively bounded depth-first search strategy. The maximum search depth must be specified interactively. Alternative solutions are generated upon request. Facilities to 'undo' user or tactic choices are provided. The resolution method itself is used (by way of bootstrapping) for correctness checking of contexts and local context introductions. Completed derivations may be added to the global context as lemmas. To this end, they must be given a name and will be available for use in subsequent queries. It is possible to 'freeze' definitions, i.e. to hide their contents and treat them as axioms. In case of clash of variable names (α-clash), unique variable names are generated by numbering. Textual editing of theories and queries is provided in the environment itself.

The special handling of contexts can be implemented efficiently. For example, translation of context elements to clauses only needs to be done once, because the main theorem guarantees that clauses remain valid in extended contexts (this is due to the fact that generalised type systems are monotonic). Verifying well-formedness of contexts can be done incrementally. Contexts can be shared amongst goals.

Figure 1: *Here, we are in the middle of an interactive session, solving the query from the example in the previous section. The second window contains the global context. It starts with a GTS specification of the Calculus of Constructions. For each goal, candidate clauses for resolution can be selected from a menu. The proof under construction is collected in the head of the derivation. Note the different local context extensions for the different subgoals.*

8.1 Technical Details

In *Constructor*, typed abstraction is denoted $[x{:}A]B$, whereas typed product is denoted as $\{x{:}A\}B$. Multiple variable introductions are permitted, e.g. $[x\ y{:}A]B$ denotes $[x{:}A][y{:}A]B$. Variables and definitions that are introduced can be declared as fix operators, much like in a Prolog fashion. For example, '[=> == [*a b*:*prop*]{*p*:*a*}*b* : {*a b*:*prop*}*prop*] *xfy 100*' declares '=>' (implication) as a right associative infix operator with priority level *100*. Application is treated as a left-associative infix operator. Bracketing is used in the usual way to over-rule priorities. Logical variables are prefixed with a '♯' symbol.

The built-in unification procedure implements a simple approximation of higher order unification with the following characteristics :

- *Higher Order Structural Matching*
  First order unification where logical variables for functors match structurally, e.g. '(♯F 0)' unifies with '(suc 0)' yielding unifier ♯F=suc.

- *Alpha conversion*
  The unification is modulo the name of bound variables, e.g. '[x:nat]x' equals '[y:nat]y'.

- *Beta conversion*
  The built in unification procedure will reduce β-redexes if necessary. To ensure that the reduction is always sound, a goal is added to ensure that the argument will have the required domain type, in the context in question.

- *Delta conversion*
  The built in unification procedure will do δ-reduction on definitions if necessary, i.e. it may expand abbreviating names.

The unification also recognizes outermost η-equality for objects, so that it can use the lambda abstraction rule to verify given application objects where the functor has not been provided with the full number of arguments. This can be regarded as the inverse operation of 'unfolding' Π-abstractions to clauses. The implemented unification procedure will always yield at most one unifier. If complicated higher order unification is required, two options are available: (1) provide appropriate auxiliary definitions to obtain the desired result (cf. *pred1* in the derivation example) (2) Interactively substitute a template of the desired proof object by hand. Checking objects (also those that can not be constructed by the unification) is always possible, because all relevant terms are known then.

The *Constructor* system automatically proves the example from the previous section without delay.

Figure 2: *Proving Tarski's Lemma [Hu87]. After automatically constructing a proof of the fixed point property for the given witness, the proof has been added to the context as a definition (named* 'Theorem'*). The proof is conducted in a version of the Calculus of Constructions. The associated completed derivation in the bottom window is only partly visible here.*

As another example, the system constructs the proof for Tarski's Lemma as formalised by [Hu87] [8] , a famous example from constructive mathematics, either by first proving the lemmas as proposed in [Hu87], or by direct automatic construction of the complete proof. The example runs within seconds. The search strategy confirms that the proofs given by Huet [Hu87] are indeed the shortest proofs. See also figure 2. As an example of a large proof (1100 lines), the system has been used to interactively construct a proof of Girard's paradox, formalising a proof in [Ba9-]. This proof was conducted in a GTS known as λ-U.

The performance of the system is good. On SUN 4 workstations, the system performs on an average 4000 unifications every second (including possible β and δ-reductions). Currently, no clause compilation takes place. Automatic construction of the proofs shown in the figures runs in seconds.

The *Constructor* system is implemented in Common Lisp. If we disregard the interface, it consists of 1800 lines of code. User licences for the *Constructor* system can be obtained at no cost. The system runs on SUN 3/4 workstations with 16+ Megabytes of memory. A full Sun Common Lisp Licence is required. A version without the window interface is also available and runs on any Common Lisp.

9 Related Systems

The Nuprl system by Constable *et al.* ([Co86]) offers an interesting and impressive interactive proof development environment that is also based on type theory. It is a significant improvement over the LCF proof system ([Go79]) that strongly influenced it. There are some inconveniences in the Nuprl system that may have been overcome by the method proposed in this document.

First of all, proof construction in Nuprl is not based on unification based clausal resolution. The inference rules are the correctness rules for the underlying type theory, and implications in a context can not be used directly as derived rules to resolve goals. Defining a new rule requires a detailed knowledge of the system and the programming language ML. Goals are posed in the empty context, and there is no notion of a theory describing the domain of interest and defining the proper axioms and inference rules. Thus, the desired context hypotheses are given as implications with the goal, and introduced later with introduction rules. As unification is not directly available, derivation of new hypotheses by instantiating others is often demanded, i.e. variables need to be given that could have been calculated.

The underlying Intuitionistic Type Theory in Nuprl is Martin-Löf's Type Theory [Ma84]. This offers a basic type structure which is richer than Coquand's, but the particular version used has as a disadvantage that proof checking is not decidable, i.e. correctness of instantiated typings cannot be verified automatically. In the Calculus of Constructions, the proof objects contain the essential information to reconstruct a meta level derivation (i.e. to check a proof), and the meta

[8] As can be seen in figure 2, this proof is done in a GTS version of CTT where [*type*:*kind*] and [*prop*:*type*], but it can also be done in $\lambda C'$.

level derivation can be thrown away upon completion. In Nuprl, tactics may inspect goals and can consist of a combination of primitive proof steps. Application of a single primitive proof step is also considered a tactic. A result of this is that derivations consist of (references to) tactics, and modifying tactics may well ruin existing proofs. Thus, in contrast with the method presented here, proof construction heuristics and proofs are inseparable.

Tactics in Nuprl are written in the meta language ML. Because referencing hypotheses is usually done with indices, and unification based resolution with hypotheses is not provided, writing tactics is difficult. Automated theorem proving has not yet been accomplished with the system ([Co86], p. 13).

One of the most powerful existing proof systems is *Isabelle* [Pa86][Pa89]. Comparison to this system is difficult, because Isabelle is a generic theorem prover, whereas the proof method proposed here is dedicated to only one single proof formalism, viz. type theory. The same remark can be made for a comparision to the work of Miller on theorem provers [Fe88].

As far as the authors know, three systems are currently under development that have similar objectives as the method proposed here, viz. assistance for proof construction in type theory. One of them is the LEGO system by Pollack (reference to be provided) in Edinburgh. First prototype implementations of this system are operational. Another very promising attempt is the ALF system, that is implemented by Nordstrom and Coquand (reference to be provided). Automated theorem proving is not yet provided in these systems, to the knowledge of the authors. Finally, Dowek [Do90] has also planned an implementation, that will use part of the approach as presented in the underlying paper. As these systems are in a preliminary stage, it is too early for a comparison.

10 Discussion

The method presented combines the advantages of resolution inference with the power of type theory.

Because proof construction is not decidable, strategic information has to be provided by users, either in the form of interactive choices, or in the form of algorithms (tactics). Resolution inference allows easy writing of tactics. The method presented has potential to be used as a logic programming language, that includes all the essential features of e.g. Prolog, but that also provides typing, higher order facilities (this implies correct handling of expressions containing binders) and the use of local assumptions, thereby creating the possibility to handle queries containing universal quantification or implication (much like λProlog [Na88]). Note that the resulting derivations are in a natural deduction style.

As a meta language, the GTS formalism is suitable to specify logical systems. The method presented makes the object level inference rules of a logic directly available for resolution instead of just the underlying correctness rules of generalised type systems, thus offering the appropriate inference level. The abbreviation mechanism provides the possibility for hiding and the use of derived lemmas.

The requirement that contexts are always grounded in derivations is essential to the method, because it avoids the central problem of unification over contexts and prevents the undesired generation of new axioms, while ensuring checkability of well-formedness and context membership. It has been demonstrated that the restriction is directly related to a fundamental problem in higher order theorem proving, and a solution is offered if completeness is desired.

It should be noted that proofs constructed by the method are in beta normal form, aside from definitions. In other words, the proofs constructed are cut-free. The proofs are not guaranteed in eta normal form, unless outermost eta reduction on objects is provided in the unification or as an explicit derivation step (as demonstrated in the derivation example of section 5).

Actual implementations of proof systems can efficiently handle many issues. For example, translation of context elements to clauses only needs to be done once, and verifying well-formedness of contexts can be done incrementally. Contexts can be shared amongst goals. A version of such a proof system, *Constructor*, equipped with a consecutively bounded depth first tactic and using a simple approximation to higher order unification, automatically constructed many non-trivial proofs, including the example proof.

Appendix A: Proof of the Main Theorem

In order to prove the main theorem, we first need the following result:

Theorem 1: Outermost η Closure [9] :

Correctness of a sequent of the form

$\Gamma \vdash [\lambda[y{:}B].\,(C\ y) : D]$, where y not free in C,

implies correctness of the sequent:

$\Gamma \vdash [C : D]$

[9]It is worth to point out, that in many generalised type systems (e.g. the AUT-QE dialect of AUTOMATH [Da80]) η-reduction is included in the system itself as a correct type conversion rule. A proof of this has not been established for generalised type systems, to the knowledge of the authors. Note that closure under η-reduction is not monotonic because η-redices from domains can occur in types, where they can not be reduced to the type of the associated η-reduced object. In CTT for example, if $[x : \Pi[u{:}\Pi[t{:}T].T].T]$ and $[y : \Pi[t{:}T].T]$ where $[T : type]$, then the object $\lambda[z : (x\ (\lambda[t{:}T].(y\ t))].\ z$ η-reduces to the object $\lambda[z{:}(x\ y)].z$, whereas the types of these objects are different.

Proof of Theorem 1

Lemma 1a:

If $\Gamma, [y : B], \Gamma' \vdash [p : P]$, where y not free in Γ' and p,
then $\exists Q$ and $\exists s$, such that

1. $P =_{\beta\delta} Q$
2. $\Gamma, \Gamma' \vdash [Q : s]$ (so y not free in Q) and $constant(s)$
3. $\Gamma, \Gamma' \vdash [p : Q]$

proof of lemma 1a: induction on derivation in $\Gamma, [y : B], \Gamma' \vdash [p : P]$ and on p.

1. $p = x$ (x is a variable)
   then $x \not\equiv y$ so $\exists T$, such that $[x : T] \in \Gamma$ or $[x : T] \in \Gamma'$, therefore $P =_{\beta\delta} T$.

2. $p = (f\ r)$
   then $\Gamma, [y : B], \Gamma' \vdash [f : \Pi[x : R].S]$
   $\Gamma, [y : B], \Gamma' \vdash [r : R]$
   and $P =_{\beta\delta} S[r/x]$.
   By (induction) hypothesis:
   $\Gamma, \Gamma' \vdash [f : \Pi[x : R'].\ S']$
   $\Gamma, \Gamma' \vdash [r : R'']$
   where $R =_{\beta\delta} R' =_{\beta\delta} R''$ and $S =_{\beta\delta} S'$
   therefore $P =_{\beta\delta} S'[r/x]$.
   It follows that $\Gamma, \Gamma' \vdash [r : R']$
   and thus $\Gamma, \Gamma' \vdash [(f\ r) : S'[r/x]]$
   hence $P =_{\beta\delta} S'[r/x]$.

3. $p = \lambda[x{:}R].\ a$
   then $P =_{\beta\delta} \Pi[x : R'].\ A$ where $R' =_{\beta\delta} R$
   and then $\Gamma, [y : B], \Gamma', [x : R] \vdash [a : A]$
   Induction gives:
   $\Gamma, \Gamma', [x : R] \vdash [a : A']$ where $A =_{\beta\delta} A'$
   and thus $\Gamma, \Gamma' \vdash [\lambda[x{:}R].\ a : \Pi[x{:}R].\ A']$
   hence $P =_{\beta\delta} \Pi[x{:}R].\ A'$.

4. $p = \Pi[x{:}R].\ q$
   then $P \equiv s$ where $constant(s)$
   and thus $\Gamma, [y : B], \Gamma', [x : R] \vdash [q : s]$
   Induction gives:
   $\Gamma, \Gamma' \vdash [q : s]$
   therefore $\Gamma, \Gamma' \vdash [\Pi[x{:}R].\ q : s]$

and $P =_{\beta\delta} s$.

This completes the proof of lemma 1a.

The assumption of Theorem 1 implies:

$D =_{\beta\delta} \Pi[y{:}B].\ D'$ and $\Gamma, [y : B] \vdash [(C\ y) : D']$

Therefore

$\Gamma, [y : B] \vdash [C : \Pi[z{:}B'].\ D'']$ where $D' =_{\beta\delta} D''[y/z]$
and $\Gamma, [y : B] \vdash [y : B']$

We now know that $B' =_{\beta\delta} B$.

According to lemma 1a:

$\Gamma \vdash [C : \Pi[z{:}B''].\ D''']$, where y not free in D''',
and $B'' =_{\beta\delta} B'$ and $D''' =_{\beta\delta} D''$

therefore $\Gamma, [y : B] \vdash [(C\ y) : D'''[y/z]]$
thus $D'''[y/z] =_{\beta\delta} D'$
and therefore $\Pi[z{:}B''].\ D''' =_{\alpha} \Pi[y{:}B''].\ D'''[y/z] =_{\beta\delta} \Pi[y{:}B].\ D' =_{\beta\delta} D$.

hence $\Gamma \vdash [C : D]$.

This completes the proof of Theorem 1.

Theorem 2 (Main Theorem):

Correctness of a sequent of the form

$\Gamma \vdash [C : \Pi[x_1{:}A_1] \ldots \Pi[x_n{:}A_n].B]$

is equivalent to the validity of the Horn clause:

$$\begin{aligned}\Gamma,\Gamma' \vdash [(C\ \sharp x_1\ ...\ \sharp x_n) : B[\sharp x_1/x_1, ..., \sharp x_n/x_n]] \Leftarrow\ & \Gamma,\Gamma' \vdash [\sharp x_1{:}A_1]\\ & \Gamma,\Gamma' \vdash [\sharp x_2{:}A_2[\sharp x_1/x_1]]\\ & \ldots\\ & \Gamma,\Gamma' \vdash [\sharp x_n{:}A_n[\sharp x_1/x_1, ..., \sharp x_{n-1}/x_{n-1}]].\end{aligned}$$

Proof

Lemma 2a: $\Gamma \vdash [C : \Pi[x{:}A].B] \Leftrightarrow \Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]]$

First $\Rightarrow$:

Assume $\Gamma \vdash [C : \Pi[x{:}A].B]$

Monotonicity guarantees that:

$\Gamma, [x'{:}A] \vdash [C : \Pi[x{:}A].B]$ (1)

The selection rule proves:

$\Gamma, [x'{:}A] \vdash [x'{:}A]$ (2)

From the application rule (rule 6) applied to (1) and (2) we can infer:

$\Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]]$

A uniform substitution of a fresh variable x' for x is performed to avoid name clashes thus ensuring a well-formed context $\Gamma, [x'{:}A]$. Discharging the assumption:

$\Gamma \vdash [C : \Pi[x{:}A].B] \Rightarrow \Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]]$ (3)

Now $\Leftarrow$:

From the lambda abstraction rule (rule 4) we derive:

$\Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]] \Rightarrow \Gamma \vdash [\lambda[x{:}A].\ (C\ x) : \Pi[x{:}A].B]$

According to Theorem 1 this implies:

$$\Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]] \Rightarrow \Gamma \vdash [C : \Pi[x{:}A].B] \quad (4)$$

Combining (3) and (4) gives:

$$\Gamma \vdash [C : \Pi[x{:}A].B] \Leftrightarrow \Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]] \quad (5)$$

This completes the proof of Lemma 2a.

By induction, it follows that:

$\Gamma \vdash [C : \Pi[x_1{:}A_1]\ ...\ \Pi[x_n{:}A_n].B] \Leftrightarrow \Gamma, [x'_1{:}A_1],\ ...\ , [x'_2{:}A_2[x'_1/x_1]],\ ...\ , [x'_n{:}A_n[x'_i/x_i]] \vdash$
$[(C\ x'_1\ ...\ x'_n) : B[x'_i/x_i]]$

To complete the proof, equivalence of the right hand side of this expression with the Horn clause

$$\begin{aligned}\Gamma, \Gamma' \vdash [(C\ \sharp x_1\ ...\ \sharp x_n) : B[\sharp x_1/x_1, ..., \sharp x_n/x_n]] \Leftarrow\ & \Gamma, \Gamma' \vdash [\sharp x_1{:}A_1]\\ & \Gamma, \Gamma' \vdash [\sharp x_2{:}A_2[\sharp x_1/x_1]]\\ & \dots\\ & \Gamma, \Gamma' \vdash [\sharp x_n{:}A_n[\sharp x_1/x_1, ..., \sharp x_{n-1}/x_{n-1}]].\end{aligned}$$

needs to be demonstrated:

Lemma 2b:
$\Gamma, [x'{:}A] \vdash [(C\ x') : B[x'/x]]$
$\Leftrightarrow$
$(\Gamma, \Gamma' \vdash [(C\ \sharp x') : B[\sharp x'/x]] \Leftarrow \Gamma, \Gamma' \vdash [\sharp x'{:}A])$

(The extension Γ' arises automatically as a result of the monotonocity of the system).

First $\Rightarrow$: directly from the substitutivity property of generalised type systems and the interpretation of Horn clauses.

Now $\Leftarrow$: by substitution of $[x'{:}A]$ for Γ'.

This completes the proof of Lemma 2b. Induction completes the proof of the main theorem.

$\square$

Appendix B: Method Extensions

This section presents extensions to the given proof construction method, that are not fundamental to the method, but that provide an essential contribution to the practical usability of a proof system. One of the extensions is a rewriting mechanism, that greatly facilitates equality reasoning, but still produces GTS proofs.

B.1 User Substitution

Direct instantiation of variables by a provided candidate term is also permitted. This rule is of course meant for interactive use, where it permits 'oracles' from the user. Moreover, for an implemented system it gives the possibility to provide terms that can not be constructed by the unification as provided by the system. Though user substitution can lead to incorrect typings (subgoals for which no derivation exists), the associated derivation is still correct with respect to its interpretation, because these subgoals can never be solved.

B.2 Multiple Queries

Although it is always possible to state problems in terms of a single goal and thus obtain a single proof object for a query, conjunctions of goals can be admitted because they are natural and allow for simultaneous querying of dependent objects.

A multiple query is thus of the form $\Gamma_1 \vdash [p_1{:}A_1]\ ...\ \Gamma_n \vdash [p_n{:}A_n]$.

A multiple derivation is different in that it has a conjunction of head terms. Proving starts off with the trivial multiple derivation:

$$\Gamma_1 \vdash [p_1{:}A_1]\ ...\ \Gamma_n \vdash [p_n{:}A_n] \Leftarrow \Gamma_1 \vdash [p_1{:}A_1]\ ...\ \Gamma_n \vdash [p_n{:}A_n]$$

Multiple queries must obey the same properties that hold for antecedents in a derivation to ensure that the necessary properties for derivations hold. When all premises have been resolved, the instantiated conclusion terms all represent facts.

Multiple queries provide a direct way to query for conjunctions and existential quantification. For example, the following multiple query provides a way to express a query for a witness $\sharp X$ for which the predicate Q holds:

$\Gamma \vdash [\sharp X{:}A]\ \ \Gamma \vdash [\sharp P{:}(Q\ \sharp X)]$

B.3 Rewriting

Equality can be formalized directly in type theory. However, the use of equality axioms in a resolution system is very inefficient, because of the combinatorial explosion of the search tree and the awkward handling of substitution. To remedy this, a term rewrite system is offered. The approach will be explained for CTT.

The following context defines Leibniz equality in CTT, with the corresponding derived theorems of reflexivity and substitutivity [10] [11] :

$$
\begin{aligned}
&[\mathit{equal} \equiv \lambda[t{:}\mathit{type}].\ \lambda[x{:}t].\ \lambda[y{:}t].\ \Pi[z{:}\Pi[p:t].\mathit{prop}].\ \Pi[q{:}(z\ x)].\ (z\ y) : \\
&\qquad \Pi[t{:}\mathit{type}].\ \Pi[x{:}t].\ \Pi[y{:}t].\ \mathit{prop}], \\
&[\mathit{refl} \equiv \lambda[t{:}\mathit{type}].\ \lambda[c{:}t].\ \Pi[z{:}\Pi[p:t].\mathit{prop}].\ \Pi[p{:}(z\ c)].\ p : \\
&\qquad \Pi[t{:}\mathit{type}].\ \Pi[x{:}t].\ (\mathit{equal}\ t\ x\ x)], \\
&[\mathit{subst} \equiv \lambda[t{:}\mathit{type}].\ \lambda[x{:}t].\ \lambda[y{:}t].\ \lambda[p:(\mathit{equal}\ t\ x\ y)]. \\
&\qquad \lambda[f:\Pi[x{:}t].\ \mathit{prop}].\ \lambda[q:(f\ x)].\ (p\ f\ q) : \\
&\qquad \Pi[t{:}\mathit{type}].\ \Pi[x{:}t].\ \Pi[y{:}t].\ \Pi[p:(\mathit{equal}\ t\ x\ y)]. \\
&\qquad \Pi[f:\Pi[x{:}t].\ \mathit{prop}].\ \Pi[q:(f\ x)].\ (f\ y)]
\end{aligned}
$$

The idea is to use correct equality assertions directly as rewrite rules. Local application of such rewrite rules results in application of the substitutivity axiom for equality. Thus, the proof constructions obtained are within the formalism of type theory and soundness is ensured.

The mechanism is best explained by a schematic example. Given a context Γ, that includes the given equality axioms, suppose that an equality assumption (or definition) is available:

$$[r : \Pi[x_1{:}A_1]\ ...\ \Pi[x_n{:}A_n].(\mathit{equal}\ t\ a\ b)] \in \Gamma$$

giving rise to the Horn clause:

$$\Gamma \vdash [r : (\mathit{equal}\ t\ a\ b)] \Leftarrow \Gamma \vdash G_1\ ...\ \Gamma \vdash G_n.$$

Suppose a derivation is under construction of the form:

[10] Instead of defining equality, it can of course be defined axiomatically in the system by omitting the definitions.

[11] The equality axioms introduced here are about equality of 3-expressions that have a 'normal' type. Also, the substitutivity axiom is given only for substitution in propositions. This stems from the fact that we distinguish explicitly between *prop*s and *type*s. Axioms for the other cases are straightforward. Note that corresponding equality axioms can also be given for 2-expressions, thus allowing a notion of equality for propositions (and types).

$$.... \Leftarrow .... \Gamma \vdash [\sharp P : T] ....$$

where T unifies with $(F\ a)$ (i.e. T is a term containing a subterm a) and $\sharp P$ is a logical variable. Using the given equality proposition as a (conditional) rewrite rule, the derivation can now be transformed to:

$$.... \Leftarrow .... \Gamma \vdash G_1 \ ...\ \Gamma \vdash G_n \ \Gamma \vdash [\sharp P' : (F\ b)] ....$$

while the other occurrences of $\sharp P$ are instantiated to the proof:

$$(subst\ t\ a\ b\ r\ F\ \sharp P').$$

Rewriting example

Let Γ_{eq} be the context consisting of the given equality axioms. Let T be the proposition [12]:

$$\Pi[t{:}type].\ \Pi[a{:}t].\ \Pi[b{:}t].\ \Pi[q{:}(equal\ t\ a\ b)].\ (equal\ t\ b\ a)$$

Suppose that in this context a proof $\sharp P$ is desired for this proposition. The associated trivial derivation is:

$$\Gamma_{eq} \vdash [\sharp P : T] \Leftarrow \Gamma_{eq} \vdash [\sharp P : T]$$

Repeated application of the lambda abstraction rule gives:

$$\Gamma_{eq} \vdash [\sharp P : T] \Leftarrow \Gamma_{eq}, \Gamma_{ext} \vdash [\sharp P' : (equal\ t\ b\ a)]$$

where Γ_{ext} denotes the context extension $[t : type], [a : t], [b : t], [q : (equal\ t\ a\ b)]$
and $\sharp P$ is instantiated to $\lambda[t{:}type].\lambda[a{:}t].\lambda[b{:}t].\lambda[q{:}(equal\ t\ a\ b)].\sharp P'$.

Rewriting with the Horn clause for the local assumption $[q : (equal\ t\ a\ b)]$ yields [13]:

[12] This asserts symmetry of equality.

[13] The substitutivity axiom allows rewriting of terms in arbitrary propositions. Here it is applied to the equality predicate itself. Note that here no extra subgoals $\Gamma \vdash G_1\ ...\ \Gamma \vdash G_n$ arise because the applied equality axiom is unconditional.

$$\Gamma_{eq} \vdash [\sharp P : T] \Leftarrow \Gamma_{eq}, \Gamma_{ext} \vdash [\sharp P'' : (equal\ t\ b\ b)]$$

instantiating $\sharp P'$ to $(subst\ t\ a\ b\ q\ \lambda[x{:}t].(equal\ b\ x)\ \sharp P'')$. Resolution with *reflexivity* now completes the proof:

$$\Gamma_{eq} \vdash [\sharp P : T] \Leftarrow$$

instantiating $\sharp P''$ to $(\mathit{refl}\ t\ b)$. The complete proof $\sharp P$ has become:

$$\lambda[t{:}type].\lambda[a{:}t].\lambda[b{:}t].\lambda[q{:}(equal\ t\ a\ b)].(subst\ t\ a\ b\ q\ \lambda[x{:}t].(equal\ b\ x)\ (\mathit{refl}\ t\ b))$$

Acknowledgements

The authors owe much gratitude to Jan Bergstra, Loe Feijs, Ton Kalker, Frank van der Linden and Rob Wieringa for numerous suggestions and corrections. Special thanks are due to Bert van Benthem Jutting, who provided the proof for outermost η-closure, to Marcel van Tien, who implemented most of *Constructor*, and to Paul Gorissen, who provided the dynamic parsing facilities for the system. Special thanks also to Henk Barendregt, for many stimulating and clarifying discussions.

References

[Ah85] Ahn, R.M.C. *Some extensions to Prolog based on* AUTOMATH. Internal Philips technical note nr. 173/85, 1985.

[Ba81] Barendregt, H. *The Lambda-Calculus: Its Syntax and Semantics.* North-Holland, 1981.

[Ba89] Barendregt, H. *Introduction to Generalised Type Systems.* Proceedings of the third Italian conference on theoretical computer science, Mantova 1989. World Scientific Publ. Also to appear in: J. Functional Programming.

[Ba90] Barendregt, H. & Hemerik. *Types in Lambda Calculi and Programming Languages.* To appear in Proceedings of European Symposium on Programming, Copenhagen, 1990.

[Ba9-] Barendregt, H. *Lambda Calculi with Types.* To appear in: *Handbook of Logic in Computer Science, Oxford University Press.* (Ed. Abramsky, S., Gabbay, D. & Maibaum, T.).

[Ba9-] Barendregt, H. & Dekkers, W. *Typed Lambda-Calculi, Syntax and Semantics.* To appear.

[Br72] De Bruijn, N.G. *Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem.* Indag. Math. 34, 5, pp 381-392, 1972.

[Br73] De Bruijn, N.G. *A survey of the project Automath.* In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms.* (Seldin & Hindley, Eds.), Academic Press, 1980.

[Co86] Constable, R.L. et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Co85] Coquand, T. *Une Theory des Constructions.* Thèse de troisième cycle, Université de Paris VII, 1985.

[Cq85] Coquand, T. & Huet, G.P. *Constructions: A Higher Order Proof System for Mechanizing Mathematics.* EUROCAL85, Linz, Springer-Verlag LNCS 203, 1985.

[Da80] van Daalen, D.T. *The Language Theory of Automath.* Ph. D. Dissertation, Eindhoven University of Technology, Dept of Mathematics, 1980.

[Do90] Dowek, G. *First Design of a Mathematical Vernacular.* Manuscript, INRIA Roquencourt, 1990.

[El89] Elliott, C.M. *Higher-order Unification with Dependent Function Types.* RTA-89, Chapel Hill, Springer-Verlag LNCS 355, 1989.

[Fe88] Felty, A. & Miller, D.A. *Specifying Theorem Provers in a Higher-Order Logic Programming Language.* CADE-9, Argonne, pp 61-80, Springer-Verlag LNCS 310, 1988.

[Fo83] Fortune, S. et al. *The Expressiveness of Simple and Second-Order Type Structures.* J. ACM 30, 1 (Jan.), pp 151-185, 1983.

[Go79] Gordon, M.J., Milner, R. & Wadsworth, C.P. *Edingburgh LCF.* Springer-Verlag LNCS 78, 1979.

[Ha87] Harper, R., Honsell, F. & Plotkin, G. *A Framework for defining logics.* Second Annual Symposium on Logic in Computer Science, Ithaca, IEEE, pp. 194-204, 1987.

[Ha89] Harper, R. & Pollack, R. *Type Checking, Universe Polymorphism, and Typical Ambiguity in the Calculus of Constructions.* Tapsoft '89, Barcelona, Springer-Verlag LNCS 352, Volume 2, 1989.

[He88] Helmink, L. & Ahn, R. *Goal Directed Proof Construction in Type Theory.* Internal Philips technical note nr. 229/88, 1988. Also available as document 28.3 of Esprit project 1222: 'Genesis'.

[He89] Helmink, L. & Tien, M. van. *Genesis Constructive Logic Machine: User's Guide.* Available as document 28.5 of Esprit project 1222: 'Genesis'.

[He90] Helmink, L. *Resolution and Type Theory.* To appear in Proceedings of European Symposium on Programming, Copenhagen, 1990.

[Hu75] Huet, G.P. *A Unification Algorithm for Typed λ-calculus.* Theoret. Comput. Sci. Vol.1, pp. 27-57, 1975.

[Hu87] Huet, G.P. *Induction Principles Formalized in the Calculus of Constructions.* Tapsoft '87, Pisa, Springer-Verlag LNCS 250, Volume 1, 1987.

[Ju76] Jutting, L.S. *A Translation of Landau's "Grundlagen" in* AUTOMATH. Ph.D. Dissertation, Eindhoven University of Technology, Dept of Mathematics, 1976.

[Ju86] Jutting, L.S. *Normalization in Coquand's system.* Internal Philips technical note nr. 156/88, 1988.

[Ma84] Martin-Löf, P. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[Na88] Nadathur G. & Miller, D.A. *An overview of λProlog.* In: *Logic Programming: Proceedings of the Fifth International Conference and Symposium.* (Kowalski & Bowen, Eds.), MIT Press, Cambridge, Massachusetts, Volume 1, pp 820-827, August 1988.

[Pa86] Paulson, L.C. *Natural Deduction as Higher-Order Resolution.* J. Logic Programming 3, pp 237-258, 1986.

[Pa89] Paulson, L.C. *The Foundation of a Generic Theorem Prover.* J. Automated Reasoning 5, pp 363-379, 1989.

[Pf89] Pfenning, F. *Elf: a language for logic definition and verified meta-programming.* Fourth Annual Symposium on Logic in Computer Science, IEEE, pp 313-322, 1989.

[Py88] Pym, D. *A unification algorithm for the logical framework.* LFCS report, Laboratory for Foundations of Computer Science, University of Edinburgh, November 1988.

[Ro65] Robinson, J.A. *A Machine Oriented Logic Based on the Resolution Principle.* J. ACM 12, 1 (Jan.), 23-49, 1965.

DEMO of the Boyer-Moore Theorem Prover and some of its extensions[1]

Matt Kaufmann[2]

Computational Logic, Inc.
Austin, Texas 78703 USA
Email: kaufmann@cli.com

The Boyer-Moore theorem prover [1, 4] is a program written by Bob Boyer and J Moore to check theorems in a quantifier-free, first-order logic that resembles Pure Lisp. The theorem prover has been used to check substantial theorems in a number of areas. It is perhaps known best for its ability to discover induction schemes automatically, a limited implementation of metatheoretic extensibility [2], and an integrated linear arithmetic procedure [3]. However, much of its utility derives from more mundane considerations such as extremely careful engineering, a simple logic, and a basic user interface that allows the user to "program" the rewriter.

An extension of the Boyer-Moore prover is PC-NQTHM ("PC" for "proof-checker", "NQTHM" being the name of the Boyer-Moore theorem prover). That system, authored by Matt Kaufmann [7, 8], extends the functionality of the Boyer-Moore prover by allowing lower-level interaction by the user as well as direction in the style of the "tactics" of systems such as LCF [12] and Nuprl [1].

In this demo I worked through a simple, but not quite trivial, example using the Boyer-Moore theorem prover and PC-NQTHM. My intention was to display both some strengths and some weaknesses of these systems, with the hope that this introduction would, although rather brief, would nevertheless give people a taste of how people really use these systems on more serious examples.

NOTE: The rest of this document is a very slight modification of [9]. It is a reasonable approximation to the demo I gave at BRA *Logical Frameworks* Workshop '90.

In this note we present a solution to a little exercise[3]: Use the Boyer-Moore theorem prover to prove that if one rotates the elements of a list by `n`, where `n` is the length of the given list, then the result is equal to the given list. In order to illustrate the use of the Boyer-Moore theorem prover (and, to a lesser extent, of its interactive enhancement PC-NQTHM[4]), we'll work through this entire example, giving lots of comments. A summary of that proof effort, i.e. the final input file, is in the first appendix. An alternate solution is presented in the second appendix.

[1]This demo description is submitted for publication in the proceedings of the BRA *Logical Frameworks* Workshop '90. It is a based heavily on Internal Note 185 of Computational Logic, Inc.

[2]This work was supported in part by ONR Contract N00014-88-C-0454. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research or the U.S. Government.

[3]first brought to my attention by Jeff Cook

[4]"PC" for "proof-checker", "NQTHM" for a name used sometimes for the Boyer-Moore prover

For a complete description of the logic of Boyer and Moore and for more information on how to use their theorem prover, see their book [4]. For more information about PC-NQTHM see [7] and [8].

Boyer and Moore's prover NQTHM can currently be obtained through anonymous ftp from cli.com. Ftp to cli.com, login as anonymous, any password, cd to /pub/nqthm, get the file README, and follow the directions. PC-NQTHM may be similarly obtained from the directory /pub/proof-checker/, also under ~ftp.

Notational conventions. We use the following notational convention from Lisp: comments begin with semicolons (;) and extend to the end of the line. The symbol '`>`' on the left margin in displays is a Lisp prompt; the s-expression following that prompt is user input, and everything up to the next prompt (or to the end of the display) is output of the system. The DEFN and PROVE-LEMMA forms that follow such prompts and are displayed in `UPPER CASE TYPEWRITER FONT` form a list of *events* that can be run successfully through the Boyer-Moore theorem prover.

Let's begin.[5] Here is a recursive definition of a function that rotates a list. Notice that this definition uses some built-in functions: `ZEROP` is true of 0 and of all objects that are not natural numbers, `CAR` returns the first member of a list, `CDR` returns all but the first member of a list, `APPEND` concatenates two lists into a single list, and `LISTP` is a recognizer for the class of pairs and non-empty lists.

```
(DEFN ROTATE (N LST)
  (IF (ZEROP N)
      LST
    (ROTATE (SUB1 N)
            (APPEND (CDR LST) (LIST (CAR LST))))))
```

The theorem prover's output is as follows. (We often omit its output henceforth.)

```
     Linear arithmetic, the lemma COUNT-NUMBERP, and the definition of ZEROP
establish that the measure (COUNT N) decreases according to the well-founded
relation LESSP in each recursive call.  Hence, ROTATE is accepted under the
principle of definition.  From the definition we can conclude that:
      (OR (LISTP (ROTATE N LST))
          (EQUAL (ROTATE N LST) LST))
is a theorem.




[ 0.1 0.1 0.0 ]
ROTATE

>
```

Here is an example, using the read-eval-print loop for the logic, of how `ROTATE` functions.

[5] Normally one would start with the initialization command `(BOOT-STRAP NQTHM)` – this is actually not necessary if one loads a saved core image.

```
>(r-loop)

Abbreviated Output Mode:  On
Type ? for help.
*(rotate 2 '(a b c d e f))
 '(C D E F A B)
*
```

Next we need the definition of the length of a list.

```
(DEFN LENGTH (X)
  (IF (LISTP X)
      (ADD1 (LENGTH (CDR X)))
    0))
```

Let's try to prove some version of our main theorem right away. Maybe we'll be lucky.

```
>(prove-lemma rotate-length ()
  (implies (listp lst)
           (equal (rotate (length lst) lst)
                  lst)))


     Name the conjecture *1.


     We will appeal to induction.  There is only one plausible induction.  We
will induct according to the following scheme:
      (AND (IMPLIES (AND (LISTP LST) (p (CDR LST)))
                    (p LST))
           (IMPLIES (NOT (LISTP LST)) (p LST))).
Linear arithmetic and the lemma CDR-LESSP inform us that the measure
(COUNT LST) decreases according to the well-founded relation LESSP in each
induction step of the scheme.  The above induction scheme produces the
following two new formulas:

Case 2. (IMPLIES (AND (NOT (LISTP (CDR LST)))
                      (LISTP LST))
                 (EQUAL (ROTATE (LENGTH LST) LST)
                        LST)).

  This simplifies, applying SUB1-ADD1, and expanding LENGTH, APPEND, and
  ROTATE, to:

        (IMPLIES (AND (NOT (LISTP (CDR LST)))
                      (LISTP LST))
```

```
                  (EQUAL (ROTATE (LENGTH (CDR LST))
                                 (LIST (CAR LST)))
                         LST)).

  This further simplifies, appealing to the lemmas CAR-CONS and CDR-CONS, and
  opening up LENGTH, EQUAL, and ROTATE, to the goal:

        (IMPLIES (AND (NOT (LISTP (CDR LST)))
                      (LISTP LST))
                 (EQUAL (LIST (CAR LST)) LST)).

  Appealing to the lemma CAR-CDR-ELIM, we now replace LST by (CONS Z X) to
  eliminate (CDR LST) and (CAR LST).  We must thus prove:

        (IMPLIES (NOT (LISTP X))
                 (EQUAL (LIST Z) (CONS Z X))).

  However this further simplifies, rewriting with CAR-CONS, CDR-CONS, and
  CONS-EQUAL, to:

        (IMPLIES (NOT (LISTP X))
                 (EQUAL NIL X)),

  which has two irrelevant terms in it.  By eliminating these terms we get the
  new formula:

      F.

Why say more?

************** F  A  I  L  E  D **************


[ 0.0 0.6 0.3 ]
NIL

>
```

Did the proof attempt fail because this is not a theorem, or is it because the Boyer-Moore theorem prover's heuristics aren't sufficient in this case? A clue is provided by the final goal printed out before the proof failed: `(IMPLIES (NOT (LISTP X)) (EQUAL NIL X))`. This says that every object that is not a non-empty list is in fact the object `nil`. But that's clearly false; consider e.g. the number 3. We can enter the read-eval-print loop for the logic to see that the theorem really does fail. In the following example, `'(a b c . d)` is a standard Lisp abbreviation for `(cons 'a (cons 'b (cons 'c 'd)))`, while `'(a b c)` is a standard Lisp abbreviation for `(cons 'a (cons 'b (cons 'c nil)))`.

```
>(r-loop)
```

```
Abbreviated Output Mode:  On
Type ? for help.
*(rotate 3 '(a b c . d))
 '(A B C)
*
```

The thing to notice above is that the innermost (last) `cons` from `'(a b c . d)`, i.e. `(cons c d)`, has a second argument that is not `nil`. So, perhaps we need some notion of "a list terminating in `NIL`."

```
(DEFN PROPERP (X)
  (IF (LISTP X)
      (PROPERP (CDR X))
    (EQUAL X NIL)))
```

One can see how this function works by tracing it.[6] Here are a couple of such traces, which we present without comment on the chance that they may be illuminating.

```
*(properp '(a b c . d))
  1> (<<PROPERP>> '((A B C . D)))
    2> (<<PROPERP>> '((B C . D)))
      3> (<<PROPERP>> '((C . D)))
        4> (<<PROPERP>> '(D))
        <4 (<<PROPERP>> F)
      <3 (<<PROPERP>> F)
    <2 (<<PROPERP>> F)
  <1 (<<PROPERP>> F)
 F
*(properp '(a b c))
  1> (<<PROPERP>> '((A B C)))
    2> (<<PROPERP>> '((B C)))
      3> (<<PROPERP>> '((C)))
        4> (<<PROPERP>> '(NIL))
        <4 (<<PROPERP>> T)
      <3 (<<PROPERP>> T)
    <2 (<<PROPERP>> T)
  <1 (<<PROPERP>> T)
 T
*
```

Now we might try the proof again. Notice that `(rotate 0 nil)` = `nil`, so we don't need a hypothesis that the list is non-empty. It's usually good to state theorems without extraneous hypotheses, so as not to confuse the prover (or the reader of its output) with irrelevant information. So, we replace the unnecessary hypothesis `(listp lst)` with the necessary one `(properp lst)`.

[6]The patch that allows such tracing is available upon request from Matt Kaufmann for use with akcl implementations of the Boyer-Moore prover.

```
(prove-lemma rotate-length ()
  (implies (properp lst)
           (equal (rotate (length lst) lst)
                  lst)))
```

Unfortunately, the proof still fails in this case. The output includes the following goals and description.

```
        (IMPLIES (PROPERP X)
                 (EQUAL (ROTATE (LENGTH X)
                                (APPEND X (LIST Z)))
                        (CONS Z (ROTATE (LENGTH X) X)))),

  which we generalize by replacing (LENGTH X) by Y.  We restrict the new
  variable by recalling the type restriction lemma noted when LENGTH was
  introduced.  This produces the new formula:

        (IMPLIES (AND (NUMBERP Y) (PROPERP X))
                 (EQUAL (ROTATE Y (APPEND X (LIST Z)))
                        (CONS Z (ROTATE Y X)))),

  which we will name *1.1.
```

We may as well abort the proof attempt at that point, since this goal is clearly not a theorem; when `Y` $= 0$ and `X` is a `PROPERP`, it says that `(APPEND X (LIST Z))` $=$ `(CONS Z X)`, which is clearly false in general. In fact generalization tends to be the least reliable heuristic in the Boyer-Moore theorem prover. But perhaps the first goal displayed above suggests (this is debatable) that we might try proving something more general, by "decoupling" the term `(LENGTH X)` from the term `X`. That is, let's consider how to rewrite `(ROTATE N X)` where `N` is any number not exceeding the length of `X`. If you think about how to prove the main theorem on paper, you may well be led to such a generalization, since it has the feel of something that might be proved by induction on `N`. Here is an informal statement of the theorem, with an informal proof by induction on `n`.

Theorem: if $n \leq k$, then

`rotate`(n, < x_1 ... x_k >) $=$ < x_{n+1} ... x_k x_1 .. x_n >

Proof sketch: by induction on n. *Base step* $n = 0$: trivial. *Inductive step.* If $0 < n \leq k$, then

`rotate(`n`, <` x_1 `...` x_k `>)`
`=` $\{$*by definition of rotate*$\}$
`rotate(`$n-1$`, <` x_2 `...` x_k x_1 `>)`
`=` $\{$*by the inductive hypothesis*$\}$
`<` x_{n+1} `...` x_k x_1 x_2 `...` x_n `>`

and we're done!

Now we formalize the operations that take the first N or last N elements of a list. (FIRSTN N LST) returns the first N elements of LST (assuming that there are at least N elements). Similarly, (NTHCDR N LST) returns the last N elements of LST (assuming that there are at least N elements).

```
(DEFN FIRSTN (N LST)
  (IF (ZEROP N)
      NIL
    (CONS (CAR LST)
          (FIRSTN (SUB1 N) (CDR LST)))))

(DEFN NTHCDR (N LST)
  (IF (ZEROP N)
      LST
    (NTHCDR (SUB1 N) (CDR LST))))
```

The big subgoal, then, is as follows.

```
(prove-lemma rotate-append ()
  (implies (and (properp lst)
                (not (lessp (length lst) n)))
           (equal (rotate n lst)
                  (append (nthcdr n lst)
                          (firstn n lst)))))
```

Having formulated this nice lemma, we are sorely tempted to hand it off to the theorem prover and hope that it is proved automatically. An alternate approach suggests thinking about the hand proof a little more and suggesting an appropriate induction scheme to the theorem prover, and we take that approach in the appendix below. But for now, let us continue in the standard manner, i.e. let us succumb to the temptation to hand off the lemma ROTATE-APPEND (that we have just stated above) to the theorem prover.

Unfortunately the proof of this lemma fails. The prover generalizes the following subgoal to one that's not a theorem.

```
(IMPLIES (AND (NUMBERP X)
              (EQUAL (ROTATE X Z)
                     (APPEND (NTHCDR X Z) (FIRSTN X Z)))
              (PROPERP Z)
              (NOT (LESSP (LENGTH Z) X)))
         (EQUAL (ROTATE X (APPEND Z (LIST V)))
                (APPEND (NTHCDR X Z)
```

```
                    (CONS V (FIRSTN X Z)))))
```

It appears that the inductive hypothesis doesn't match up very well with the induction conclusion, since the conclusion mentions the notion of `ROTATE`-ing an `APPEND` term, `(ROTATE X (APPEND Z (LIST V)))`. So let us "generalize" the theorem we are trying to prove by considering what happens when we append extra stuff to the end of the list before rotating it.

```
(prove-lemma rotate-append ()
  (implies (and (properp lst)
                (not (lessp (length lst) n)))
           (equal (rotate n (append lst extra))
                  (append (nthcdr n lst)
                          (append extra (firstn n lst))))))
```

This lemma's proof was not successful – it generated the unprovable subgoal

```
     (EQUAL EXTRA (APPEND EXTRA NIL)).
```

Inspection suggests that we need `(PROPERP EXTRA)` in order for the `N` = 0 case to be true, which makes sense since then `(EQUAL EXTRA (APPEND EXTRA NIL))` is true. But with that added hypothesis it isn't too hard to see that we no longer need to assume `(PROPERP LST)`.

Thus our next try is as follows.

```
(prove-lemma rotate-append ()
  (implies (and (properp extra)
                (not (lessp (length lst) n)))
           (equal (rotate n (append lst extra))
                  (append (nthcdr n lst)
                          (append extra (firstn n lst))))))
```

However, the proof still fails. Inspection of the output reveals the term `(APPEND (APPEND Z EXTRA) (LIST V))`, which suggests (at least to an experienced user) that we should prove the associativity of `APPEND`. Some day it will be the case that Boyer-Moore theorem prover users always load libraries with such basic facts built in. Anyhow, let's prove that lemma now before looking any further at the failed proof of ROTATE-APPEND.

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND (REWRITE)
  (EQUAL (APPEND (APPEND X Y) Z)
         (APPEND X (APPEND Y Z))))
```

We may as well also prove the aforementioned fact about `APPEND`ing to `NIL`.

```
(PROVE-LEMMA APPEND-NIL (REWRITE)
  (IMPLIES (PROPERP X)
           (EQUAL (APPEND X NIL)
                  X)))
```

Now we try again to prove ROTATE-APPEND. The proof still fails. Again the theorem prover chooses to generalize. The goal printed out just before generalization is hard to understand (in my opinion, anyhow). So, I'll enter the "proof-checker" enhancement of the Boyer-Moore prover, PC-NQTHM. The VERIFY form below causes us to enter PC-NQTHM with the indicated goal, which is the statement of the theorem ROTATE-APPEND.

```
>(verify (implies (and (properp extra)
                       (not (lessp (length lst) n)))
                  (equal (rotate n (append lst extra))
                         (append (nthcdr n lst)
                                 (append extra (firstn n lst))))))

;; All input below follows the PC-NQTHM prompt, `->: '; the rest (except for
;; comments which extend on a given line from a semicolon to the end of the
;; line) is output from the system.
->: p ;; Print the ``current term''.

(IMPLIES (AND (PROPERP EXTRA)
              (NOT (LESSP (LENGTH LST) N)))
         (EQUAL (ROTATE N (APPEND LST EXTRA))
                (APPEND (NTHCDR N LST)
                        (APPEND EXTRA (FIRSTN N LST)))))
->: induct ;; Divide into subgoals using a heuristically-chosen induction scheme.


Inducting according to the scheme:

      (AND (IMPLIES (OR (EQUAL N 0) (NOT (NUMBERP N)))
                    (p EXTRA LST N))
           (IMPLIES (AND (NOT (OR (EQUAL N 0) (NOT (NUMBERP N))))
                         (OR (EQUAL (LENGTH LST) 0)
                             (NOT (NUMBERP (LENGTH LST)))))
                    (p EXTRA LST N))
           (IMPLIES (AND (NOT (OR (EQUAL N 0) (NOT (NUMBERP N))))
                         (NOT (OR (EQUAL (LENGTH LST) 0)
                                  (NOT (NUMBERP (LENGTH LST)))))
                         (p EXTRA (CDR LST) (SUB1 N)))
                    (p EXTRA LST N)))

Creating 3 new subgoals, (MAIN . 1), (MAIN . 2), and (MAIN . 3).

The proof of the current goal, MAIN, has been completed.  However, the
```

```
following subgoals of MAIN remain to be proved: (MAIN . 1), (MAIN . 2), and
(MAIN . 3).
Now proving (MAIN . 1).
->: p

(IMPLIES (OR (EQUAL N 0) (NOT (NUMBERP N)))
         (IMPLIES (AND (PROPERP EXTRA)
                       (NOT (LESSP (LENGTH LST) N)))
                  (EQUAL (ROTATE N (APPEND LST EXTRA))
                         (APPEND (NTHCDR N LST)
                                 (APPEND EXTRA (FIRSTN N LST))))))
->: prove
;; We call the Boyer-Moore theorem prover, since this looks like an easy base case.


***** Now entering the theorem prover *****:

This simplifies, applying APPEND-NIL, and opening up NOT, OR, EQUAL, LESSP,
ROTATE, NTHCDR, and FIRSTN, to:

      T.

Q.E.D.


The current goal, (MAIN . 1), has been proved, and has no dependents.
Now proving (MAIN . 2).
->: p

(IMPLIES (AND (NOT (EQUAL N 0))
              (NUMBERP N)
              (OR (EQUAL (LENGTH LST) 0)
                  (NOT (NUMBERP (LENGTH LST)))))
         (IMPLIES (AND (PROPERP EXTRA)
                       (NOT (LESSP (LENGTH LST) N)))
                  (EQUAL (ROTATE N (APPEND LST EXTRA))
                         (APPEND (NTHCDR N LST)
                                 (APPEND EXTRA (FIRSTN N LST))))))
->: prove
;; We call the prover again, since this also looks like an easy base case.


***** Now entering the theorem prover *****:

This formula simplifies, opening up NOT, OR, EQUAL, and LESSP, to:

      T.

Q.E.D.
```

```
The current goal, (MAIN . 2), has been proved, and has no dependents.
Now proving (MAIN . 3).
->: p

(IMPLIES (AND (NOT (EQUAL N 0))
              (NUMBERP N)
              (NOT (EQUAL (LENGTH LST) 0))
              (NUMBERP (LENGTH LST))
              (IMPLIES (AND (PROPERP EXTRA)
                            (NOT (LESSP (LENGTH (CDR LST)) (SUB1 N))))
                       (EQUAL (ROTATE (SUB1 N)
                                      (APPEND (CDR LST) EXTRA))
                              (APPEND (NTHCDR (SUB1 N) (CDR LST))
                                      (APPEND EXTRA
                                              (FIRSTN (SUB1 N) (CDR LST)))))))
         (IMPLIES (AND (PROPERP EXTRA)
                       (NOT (LESSP (LENGTH LST) N)))
                  (EQUAL (ROTATE N (APPEND LST EXTRA))
                         (APPEND (NTHCDR N LST)
                                 (APPEND EXTRA (FIRSTN N LST))))))
->: split
;; Split into subgoals based on the propositional structure of the goal.


Creating 2 new subgoals, ((MAIN . 3) . 1) and ((MAIN . 3) . 2).

The proof of the current goal, (MAIN . 3), has been completed.  However, the
following subgoals of (MAIN . 3) remain to be proved: ((MAIN . 3) . 1) and
((MAIN . 3) . 2).
Now proving ((MAIN . 3) . 1).
->: th  ;; Mnemonic for ``theorem'' -- display the current goal.

*** Active top-level hypotheses:
H1.  (NOT (EQUAL N 0))
H2.  (NUMBERP N)
H3.  (NOT (EQUAL (LENGTH LST) 0))
H4.  (NUMBERP (LENGTH LST))
H5.  (EQUAL (ROTATE (SUB1 N)
                    (APPEND (CDR LST) EXTRA))
            (APPEND (NTHCDR (SUB1 N) (CDR LST))
                    (APPEND EXTRA
                            (FIRSTN (SUB1 N) (CDR LST)))))
H6.  (PROPERP EXTRA)
H7.  (NOT (LESSP (LENGTH LST) N))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (ROTATE N (APPEND LST EXTRA))
       (APPEND (NTHCDR N LST)
```

```
               (APPEND EXTRA (FIRSTN N LST))))
->: goals  ;; List the names of the unproved goals, top one first.

((MAIN . 3) . 1)
((MAIN . 3) . 2)
->: change-goal  ;; Change to the other goal; maybe it's easier!

Now proving ((MAIN . 3) . 2).
->: goals  ;; Notice that the goals have indeed been switched.

((MAIN . 3) . 2)
((MAIN . 3) . 1)
->: th  ;; Mnemonic for ''theorem'' -- display the current goal.

*** Active top-level hypotheses:
H1.  (NOT (EQUAL N 0))
H2.  (NUMBERP N)
H3.  (NOT (EQUAL (LENGTH LST) 0))
H4.  (NUMBERP (LENGTH LST))
H5.  (LESSP (LENGTH (CDR LST)) (SUB1 N))
H6.  (PROPERP EXTRA)
H7.  (NOT (LESSP (LENGTH LST) N))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (ROTATE N (APPEND LST EXTRA))
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: prove  ;; Hypotheses 5 and 6 are contradictory.


***** Now entering the theorem prover *****:

This simplifies, applying the lemma SUB1-ADD1, and unfolding LENGTH and LESSP,
to:

      T.

Q.E.D.


The current goal, ((MAIN . 3) . 2), has been proved, and has no dependents.
Now proving ((MAIN . 3) . 1).
->: goals  ;; Only one goal remains, but it's the hard one.

((MAIN . 3) . 1)
->: th

*** Active top-level hypotheses:
```

```
H1.  (NOT (EQUAL N 0))
H2.  (NUMBERP N)
H3.  (NOT (EQUAL (LENGTH LST) 0))
H4.  (NUMBERP (LENGTH LST))
H5.  (EQUAL (ROTATE (SUB1 N)
                    (APPEND (CDR LST) EXTRA))
             (APPEND (NTHCDR (SUB1 N) (CDR LST))
                     (APPEND EXTRA
                             (FIRSTN (SUB1 N) (CDR LST)))))
H6.  (PROPERP EXTRA)
H7.  (NOT (LESSP (LENGTH LST) N))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (ROTATE N (APPEND LST EXTRA))
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: (dive 1 2)
;; Focus on the subterm obtained by diving to the first argument of
;; the EQUAL term and then to the second argument of the ROTATE term.

->: p  ;; Notice what the current subterm is now.

(APPEND LST EXTRA)
->: x  ;; Expand this call of APPEND and simplify.
->: p

(IF (LISTP LST)
    (CONS (CAR LST)
          (APPEND (CDR LST) EXTRA))
    EXTRA)
->: comm
;; Since the term (LISTP LST) didn't simplify to T (true), we might want to
;; back up and claim it to hold.  But first we might wish to recall where we are.

The top-level commands thus far (in reverse order, i.e. last one first) have
been:
1. X
2. (DIVE 1 2)
3. PROVE
4. CHANGE-GOAL
5. SPLIT
6. PROVE
7. PROVE
8. INDUCT
9. START
->: (undo 2)  ;; Pop off the two top proof states.

Last instruction undone: (DIVE 1 2).
```

```
->: th

*** Active top-level hypotheses:
H1.  (NOT (EQUAL N 0))
H2.  (NUMBERP N)
H3.  (NOT (EQUAL (LENGTH LST) 0))
H4.  (NUMBERP (LENGTH LST))
H5.  (EQUAL (ROTATE (SUB1 N)
                    (APPEND (CDR LST) EXTRA))
            (APPEND (NTHCDR (SUB1 N) (CDR LST))
                    (APPEND EXTRA
                            (FIRSTN (SUB1 N) (CDR LST)))))
H6.  (PROPERP EXTRA)
H7.  (NOT (LESSP (LENGTH LST) N))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (ROTATE N (APPEND LST EXTRA))
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: (claim (listp lst))
;; The theorem prover should be able to prove this, given hypothesis H3.


***** Now entering the theorem prover *****:

This formula simplifies, unfolding LENGTH and EQUAL, to:

      T.

Q.E.D.

->: (dive 1 2)  ;; same explanation as before

->: p

(APPEND LST EXTRA)
->: x  ;; same explanation as before

->: p
  ;; Notice that this time the ``test'' (LISTP LST) of the IF term simplified to T.

(CONS (CAR LST)
      (APPEND (CDR LST) EXTRA))
->: pp-top
  ;; Pretty-print from the top of the conclusion, highlighting the current term.

(EQUAL (ROTATE N
               (*** (CONS (CAR LST)
```

```
                         (APPEND (CDR LST) EXTRA))
                   ***))
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: up  ;; Move up one level, so that we can expand the call of ROTATE.

->: pp-top

(EQUAL (*** (ROTATE N
                   (CONS (CAR LST)
                         (APPEND (CDR LST) EXTRA)))
           ***)
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: x

->: p

(ROTATE (SUB1 N)
        (APPEND (APPEND (CDR LST) EXTRA)
                (LIST (CAR LST))))
->: top
;; Move to the top of the conclusion, so that the current term is the entire conclusion.

->: th

*** Active top-level hypotheses:
H1.  (NOT (EQUAL N 0))
H2.  (NUMBERP N)
H3.  (NOT (EQUAL (LENGTH LST) 0))
H4.  (NUMBERP (LENGTH LST))
H5.  (EQUAL (ROTATE (SUB1 N)
                   (APPEND (CDR LST) EXTRA))
            (APPEND (NTHCDR (SUB1 N) (CDR LST))
                   (APPEND EXTRA
                           (FIRSTN (SUB1 N) (CDR LST)))))
H6.  (PROPERP EXTRA)
H7.  (NOT (LESSP (LENGTH LST) N))
H8.  (LISTP LST)

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (ROTATE (SUB1 N)
               (APPEND (APPEND (CDR LST) EXTRA)
                       (LIST (CAR LST))))
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: (s lemmas)
;; Simplify using rewrite rules, which include the associativity of APPEND.
```

```
->: th

*** Active top-level hypotheses:
H1.  (NOT (EQUAL N 0))
H2.  (NUMBERP N)
H3.  (NOT (EQUAL (LENGTH LST) 0))
H4.  (NUMBERP (LENGTH LST))
H5.  (EQUAL (ROTATE (SUB1 N)
                    (APPEND (CDR LST) EXTRA))
            (APPEND (NTHCDR (SUB1 N) (CDR LST))
                    (APPEND EXTRA
                            (FIRSTN (SUB1 N) (CDR LST)))))
H6.  (PROPERP EXTRA)
H7.  (NOT (LESSP (LENGTH LST) N))
H8.  (LISTP LST)

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (ROTATE (SUB1 N)
               (APPEND (CDR LST)
                       (APPEND EXTRA (LIST (CAR LST)))))
       (APPEND (NTHCDR N LST)
               (APPEND EXTRA (FIRSTN N LST))))
->: exit
;; Let's quit, now that we see that the inductive hypothesis (H5)
;; isn't quite what we want.


Quitting the interactive proof checker.  Submit (VERIFY) to get back in at
this state.  **NOTE** -- No event has been stored.
NIL

>
```

Recall the induction scheme printed at the outset of the proof above:

```
(AND (IMPLIES (OR (EQUAL N 0) (NOT (NUMBERP N)))
              (p EXTRA LST N))
     (IMPLIES (AND (NOT (OR (EQUAL N 0) (NOT (NUMBERP N))))
                   (OR (EQUAL (LENGTH LST) 0)
                       (NOT (NUMBERP (LENGTH LST)))))
              (p EXTRA LST N))
     (IMPLIES (AND (NOT (OR (EQUAL N 0) (NOT (NUMBERP N))))
                   (NOT (OR (EQUAL (LENGTH LST) 0)
                            (NOT (NUMBERP (LENGTH LST)))))
                   (p EXTRA (CDR LST) (SUB1 N)))
```

```
             (p EXTRA LST N)))
```

The interactive proof attempt shows that we should perhaps modify this scheme slightly. We indicate the changes in lower case below.

```
(AND (IMPLIES (OR (EQUAL N 0) (NOT (NUMBERP N)))
              (p EXTRA LST N))
     (IMPLIES (AND (NOT (OR (EQUAL N 0) (NOT (NUMBERP N))))
                   (OR (EQUAL (LENGTH LST) 0)
                       (NOT (NUMBERP (LENGTH LST)))))
              (p EXTRA LST N))
     (IMPLIES (AND (NOT (OR (EQUAL N 0) (NOT (NUMBERP N))))
                   (NOT (OR (EQUAL (LENGTH LST) 0)
                            (NOT (NUMBERP (LENGTH LST)))))
                   (p (append EXTRA (list (car lst)))
                      (CDR LST) (SUB1 N)))
              (p EXTRA LST N)))
```

How do we instruct the theorem prover to induct in the manner suggested above? We define a function whose recursion is "analogous" to this scheme, as the following example will illustrate. Later we'll give a hint to the theorem prover to use the induction scheme suggested by this definition.

```
(DEFN ROTATE-APPEND-INDUCTION (N LST EXTRA)
  (IF (OR (EQUAL N 0) (NOT (NUMBERP N)))
      T
    (IF (NLISTP LST)
        T
      (ROTATE-APPEND-INDUCTION
       (SUB1 N)
       (CDR LST)
       (APPEND EXTRA (LIST (CAR LST)))))))
```

Another attempt at proving ROTATE-APPEND now leads to a term of the form

```
(IMPLIES (AND (NUMBERP V)
              (NOT (PROPERP (APPEND EXTRA (LIST X))))
              (PROPERP EXTRA)
              (NOT (LESSP (LENGTH Z) V)))
         .....)
```

which is absurd since the second hypothesis must fail. So we ask the theorem prover to prove the following rewrite rule in order to help it with the proof of ROTATE-APPEND.

```
(PROVE-LEMMA PROPERP-APPEND (REWRITE)
  (EQUAL (PROPERP (APPEND X Y))
         (PROPERP Y)))
```

Now the proof of ROTATE-APPEND (with an appropriate hint to induct as in the definition of `ROTATE-APPEND-INDUCTION`) succeeds. That's because the problematic term `(NOT (PROPERP (APPEND EXTRA (LIST X))))`, discussed above, is rewritten to `(NOT (PROPERP (LIST X)))` (using the rule PROPERP-APPEND that is displayed immediately above), and the theorem prover's simplifier can simplify this to `F` (*false*). Notice that there is no point in making ROTATE-APPEND a rewrite rule, since we'll just instantiate it in order to get the the main theorem ROTATE-LENGTH proved. (The `APPEND` term in `(ROTATE N (APPEND LST EXTRA))` prevents it from being used automatically in the proof of ROTATE-LENGTH.)

```
(PROVE-LEMMA ROTATE-APPEND ()
  (IMPLIES (AND (PROPERP EXTRA)
                (NOT (LESSP (LENGTH LST) N)))
           (EQUAL (ROTATE N (APPEND LST EXTRA))
                  (APPEND (NTHCDR N LST)
                          (APPEND EXTRA (FIRSTN N LST)))))
  ((INDUCT (ROTATE-APPEND-INDUCTION N LST EXTRA))))
```

Now we can try to get the theorem prover to prove the main theorem, by giving it an appropriate hint. Unfortunately, there seems to be some problem, as we now see.

```
>(prove-lemma rotate-length ()
  (implies (properp lst)
           (equal (rotate (length lst) lst)
                  lst))
  ((use (rotate-append (extra nil) (n (length lst))))))

This conjecture simplifies, applying the lemma APPEND-NIL, and expanding the
definitions of PROPERP, NOT, AND, LISTP, APPEND, and IMPLIES, to the following
two new formulas:

Case 2. (IMPLIES (AND (LESSP (LENGTH LST) (LENGTH LST))
                      (PROPERP LST))
                 (EQUAL (ROTATE (LENGTH LST) LST)
                        LST)).

  However this again simplifies, using linear arithmetic, to:

        T.

Case 1. (IMPLIES (AND (EQUAL (ROTATE (LENGTH LST) LST)
                             (APPEND (NTHCDR (LENGTH LST) LST)
                                     (FIRSTN (LENGTH LST) LST)))
                      (PROPERP LST))
```

```
                (EQUAL (ROTATE (LENGTH LST) LST)
                       LST)).

  We use the above equality hypothesis by substituting:
        (APPEND (NTHCDR (LENGTH LST) LST)
                (FIRSTN (LENGTH LST) LST))
  for (ROTATE (LENGTH LST) LST) and keeping the equality hypothesis.  The
  result is:

        (IMPLIES (AND (EQUAL (ROTATE (LENGTH LST) LST)
                             (APPEND (NTHCDR (LENGTH LST) LST)
                                     (FIRSTN (LENGTH LST) LST)))
                      (PROPERP LST))
                 (EQUAL (APPEND (NTHCDR (LENGTH LST) LST)
                                (FIRSTN (LENGTH LST) LST))
                        LST)).

  This further simplifies, clearly, to:

        (IMPLIES (AND (EQUAL (ROTATE (LENGTH LST) LST)
                             (APPEND (NTHCDR (LENGTH LST) LST)
                                     (FIRSTN (LENGTH LST) LST)))
                      (PROPERP LST))
                 (EQUAL (ROTATE (LENGTH LST) LST)
                        LST)),

  which we would normally push and work on later by induction.  But if we must
  use induction to prove the input conjecture, we prefer to induct on the
  original formulation of the problem.  Thus we will disregard all that we
  have previously done, give the name *1 to the original input, and work on
  it.

[.... proof aborted by me ....]
```

The goal shown immediately above would be sufficient if only the theorem prover knew the following two obvious facts, which we'll ask it to prove and to store as rewrite rules. These can be proved in less than a second each on a Sun3/60.

```
(PROVE-LEMMA NTHCDR-LENGTH (REWRITE)
  (IMPLIES (PROPERP LST)
           (EQUAL (NTHCDR (LENGTH LST) LST)
                  NIL)))

(PROVE-LEMMA FIRSTN-LENGTH (REWRITE)
  (IMPLIES (PROPERP LST)
           (EQUAL (FIRSTN (LENGTH LST) LST)
                  LST)))
```

Finally the theorem prover is able to prove our main theorem.

```
(PROVE-LEMMA ROTATE-LENGTH ()
  (IMPLIES (PROPERP LST)
           (EQUAL (ROTATE (LENGTH LST) LST)
                  LST))
  ((USE (ROTATE-APPEND (EXTRA NIL) (N (LENGTH LST))))))
```

APPENDIX 1: LIST OF EVENTS

Here is the sequence of events presented above.

```
(DEFN ROTATE (N LST)
  (IF (ZEROP N)
      LST
    (ROTATE (SUB1 N)
            (APPEND (CDR LST) (LIST (CAR LST))))))

(DEFN LENGTH (X)
  (IF (LISTP X)
      (ADD1 (LENGTH (CDR X)))
    0))

(DEFN PROPERP (X)
  (IF (LISTP X)
      (PROPERP (CDR X))
    (EQUAL X NIL)))

(DEFN FIRSTN (N LST)
  (IF (ZEROP N)
      NIL
    (CONS (CAR LST)
          (FIRSTN (SUB1 N) (CDR LST)))))

(DEFN NTHCDR (N LST)
  (IF (ZEROP N)
      LST
    (NTHCDR (SUB1 N) (CDR LST))))

(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND (REWRITE)
  (EQUAL (APPEND (APPEND X Y) Z)
         (APPEND X (APPEND Y Z))))

(PROVE-LEMMA APPEND-NIL (REWRITE)
```

```
  (IMPLIES (PROPERP X)
           (EQUAL (APPEND X NIL)
                  X)))

(DEFN ROTATE-APPEND-INDUCTION (N LST EXTRA)
  (IF (OR (EQUAL N 0) (NOT (NUMBERP N)))
      T
    (IF (NLISTP LST)
        T
      (ROTATE-APPEND-INDUCTION
       (SUB1 N)
       (CDR LST)
       (APPEND EXTRA (LIST (CAR LST)))))))

(PROVE-LEMMA PROPERP-APPEND (REWRITE)
  (EQUAL (PROPERP (APPEND X Y))
         (PROPERP Y)))

(PROVE-LEMMA ROTATE-APPEND ()
  (IMPLIES (AND (PROPERP EXTRA)
                (NOT (LESSP (LENGTH LST) N)))
           (EQUAL (ROTATE N (APPEND LST EXTRA))
                  (APPEND (NTHCDR N LST)
                          (APPEND EXTRA (FIRSTN N LST)))))
  ((INDUCT (ROTATE-APPEND-INDUCTION N LST EXTRA))))

(PROVE-LEMMA NTHCDR-LENGTH (REWRITE)
  (IMPLIES (PROPERP LST)
           (EQUAL (NTHCDR (LENGTH LST) LST)
                  NIL)))

(PROVE-LEMMA FIRSTN-LENGTH (REWRITE)
  (IMPLIES (PROPERP LST)
           (EQUAL (FIRSTN (LENGTH LST) LST)
                  LST)))

(PROVE-LEMMA ROTATE-LENGTH ()
  (IMPLIES (PROPERP LST)
           (EQUAL (ROTATE (LENGTH LST) LST)
                  LST))
  ((USE (ROTATE-APPEND (EXTRA NIL) (N (LENGTH LST))))))
```

APPENDIX 2: AN ALTERNATE APPROACH

Here is a list of events that suggests an alternate approach to the proof, in which we do not consider what happens when one rotates the `APPEND` of a list with some "extra" list. The definitions of ROTATE, LENGTH, PROPERP, FIRSTN, and NTHCDR are the same as in the proof described above, as are the statements of the lemmas PROPERP-APPEND, APPEND-NIL, ASSOCIATIVITY-OF-APPEND, NTHCDR-LENGTH, and FIRSTN-LENGTH. However, we include those too, for completeness. Note: a replay of these 16 events on a Sun3/60 with 20 megabytes main memory took just over a half minute (real time), while a replay of the original 13 events described in the text above took about 20 seconds real time.

```
(DEFN ROTATE (N LST)  ;; same as before
  (IF (ZEROP N)
      LST
      (ROTATE (SUB1 N)
              (APPEND (CDR LST) (LIST (CAR LST))))))

(DEFN LENGTH (X)  ;; same as before
  (IF (LISTP X)
      (ADD1 (LENGTH (CDR X)))
      0))

(DEFN PROPERP (X)  ;; same as before
  (IF (LISTP X)
      (PROPERP (CDR X))
      (EQUAL X NIL)))

(DEFN FIRSTN (N LST)  ;; same as before
  (IF (ZEROP N)
      NIL
      (CONS (CAR LST)
            (FIRSTN (SUB1 N) (CDR LST)))))

(DEFN NTHCDR (N LST)  ;; same as before
  (IF (ZEROP N)
      LST
      (NTHCDR (SUB1 N) (CDR LST))))

(DEFN ROTATE-INDUCTION (N LST)
  ;; This function suggests the induction carried out in the informal proof above.
  (IF (ZEROP N)
      T
      (ROTATE-INDUCTION (SUB1 N)
                        (APPEND (CDR LST) (LIST (CAR LST))))))

(PROVE-LEMMA PROPERP-APPEND (REWRITE)  ;; same as before
  (EQUAL (PROPERP (APPEND X Y))
         (PROPERP Y)))

(PROVE-LEMMA APPEND-NIL (REWRITE)  ;; same as before
  (IMPLIES (PROPERP X)
           (EQUAL (APPEND X NIL) X)))
```

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND (REWRITE)  ;; same as before
  (EQUAL (APPEND (APPEND X Y) Z)
         (APPEND X (APPEND Y Z))))

;; The need for the following 3 lemmas was suggested by examining failed proofs
;; of ROTATE-LENGTH-LEMMA.

(PROVE-LEMMA LENGTH-APPEND (REWRITE)
  (EQUAL (LENGTH (APPEND X Y))
         (PLUS (LENGTH X) (LENGTH Y))))

(PROVE-LEMMA LEMMA-1 (REWRITE)
  (IMPLIES (NOT (LESSP (LENGTH Z) N))
           (EQUAL (NTHCDR N (APPEND Z Y))
                  (APPEND (NTHCDR N Z) Y))))

(PROVE-LEMMA LEMMA-2 (REWRITE)
  (IMPLIES (NOT (LESSP (LENGTH Z) X))
           (EQUAL (FIRSTN X (APPEND Z Y))
                  (FIRSTN X Z))))

(PROVE-LEMMA ROTATE-LENGTH-LEMMA (REWRITE)
  (IMPLIES (AND (PROPERP LST)
                (NOT (LESSP (LENGTH LST) N)))
           (EQUAL (ROTATE N LST)
                  (APPEND (NTHCDR N LST)
                          (FIRSTN N LST))))
  ((INDUCT (ROTATE-INDUCTION N LST))))

(PROVE-LEMMA NTHCDR-LENGTH (REWRITE)  ;; same as before
  (IMPLIES (PROPERP LST)
           (EQUAL (NTHCDR (LENGTH LST) LST)
                  NIL)))

(PROVE-LEMMA FIRSTN-LENGTH (REWRITE)  ;; same as before
  (IMPLIES (PROPERP LST)
           (EQUAL (FIRSTN (LENGTH LST) LST)
                  LST)))

(PROVE-LEMMA ROTATE-LENGTH NIL  ;; same as before
  (IMPLIES (PROPERP LST)
           (EQUAL (ROTATE (LENGTH LST) LST)
                  LST)))
```

References

[1] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979.

[2] Robert S. Boyer and J Strother Moore. "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures." In *The Correctness Problem in Computer Science*, ed. Robert S. Boyer and J Strother Moore, Academic Press, London, 1981.

[3] Robert S. Boyer and J Strother Moore. *Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic.* Technical Report ICSCA-CMP-44, University of Texas at Austin, 1985.

[4] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[5] R.L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[6] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[7] Matt Kaufmann. *A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover.* Technical Report CLI-19, Computational Logic, Inc., May 1988.

[8] Matt Kaufmann. *Addition of Free Variables to an Interactive Enchancement of the Boyer-Moore Theorem Prover.* Technical Report CLI-42, Computational Logic, Inc., 1990.

[9] Matt Kaufmann. *An Instructive Example for Beginning Users of the Boyer-Moore Theorem Prover.* Internal Note 185, Computational Logic, Inc., April 1990.

An Extension to ML to Handle Bound Variables in Data Structures: Preliminary Report

Dale Miller
Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104–6389 USA
dale@cis.upenn.edu

Abstract

Most conventional programming languages have direct methods for representing first-order terms (say, via concrete datatypes in ML). If it is necessary to represent structures containing bound variables, such as λ-terms, formulas, types, or proofs, these must first be mapped into first-order terms, and then a significant number of auxiliary procedures must be implemented to manage bound variable names, check for free occurrences, do substitution, test for equality modulo alpha-conversion, etc. We shall show how the applicative core of the ML programming language can be enhanced so that λ-terms can be represented more directly and so that the enhanced language, called ML_λ, provides a more elegant method of manipulating bound variables within data structures. In fact, the names of bound variables will not be accessible to the ML_λ programmer. This extension to ML involves the following: introduction of the new type constructor `'a => 'b` for the type of λ-terms formed by abstracting a parameter of type `'a` out of a term of type `'b`; a very restricted and simple form of higher-order pattern matching; a method for extending a given data structure with a new constructor; and, a method for extending function definitions to handle such new constructors. We present several examples of ML_λ programs.

1 Introduction

Recent work in the specification of a wide variety of meta-programming systems — type checkers and inferrers, theorem provers, program manipulation systems, evaluators, and compilers — has revealed several important specification techniques, including one called *higher-order abstract syntax* [16]. This specification technique uses a typed λ-calculus to succinctly capture many complex syntactic notions pertaining to data structures containing notions of bound variables, such as those of free and bound occurrences, scopes of binders, equality up to alphabetic change of bound variable names, and substitutions. Huet and Lang [9] seem to have been the first to recognize the potential of this approach to abstract syntax in an actual implementation. Some of their ideas were later generalized in [11] to a logic programming setting. Higher-order abstract syntax is now central to

[1]This research was supported in part by grants ONR N00014-88-K-0633, NSF CCR-87-05596, and DARPA N00014-85-K-0018. I am grateful to John Hannan, Frank Pfenning, and several attendees of the Logical Frameworks BRA Workshop for comments on the content of this paper.

several recent specification systems. The Logical Framework (LF) [8], the Calculus of Constructions (CC) [3], and hereditary Harrop formulas [12] are some recent logics that support this new view of syntax. These systems have been used to specify various meta-programming tasks: LF has been used to specify numerous proof systems [1] as well as various aspects of conventional programming language semantics [2]; hereditary Harrop formulas have been implemented in the Isabelle theorem prover [14, 15] and in λProlog, where a wide range of meta-programs have been written [4, 5, 6]; and, CC has been used to specify and compute with several deep mathematical theorems as well as specify and develop various algorithms.

Such a specification technique has been accounted for in practice in essentially two ways. The first can be called the package-style approach. That is, an implementor writes a collection of functions and routines in a given programming language that captures this specification technique. For example, the Mentor system [9] contained an implementation of second-order matching in order to implement template matching for sophisticated program transformations. The Isabelle system is a package of ML programs that can manipulate the higher-order abstract syntax of various object logics.

The second approach to accounting for these specification techniques is to design programming languages in which they are directly incorporated. The programming languages λProlog [13] and Elf [17] are two such programming languages.

In all of these cases — Mentor, Isabelle, λProlog, and Elf — rather strong forms of unification have been used to manipulate typed λ-terms. Thus, at first glance, it would seem difficult to find an extension to the functional programming language ML that could incorporate higher-order abstract syntax since general unification, especially unification that may be undecidable and does not admit most general unifiers, is difficult to integrate directly into a functional setting.

In this paper, we shall attempt to illustrate how ML can be extended so as to allow the direct manipulation of structures with bound variables. The extensions, called ML_λ, will therefore permit direct exploitation of the technique of higher-order abstract syntax. The key idea to making this extension is that a weak form of unification of simply typed λ-terms, described in [10], is adequate for capturing much of the ideas of higher-order abstract syntax and that if unification is restricted to pattern matching, it comprises a simple and natural extension to usual ML pattern matching.

Familiarity with the basic elements of ML is necessary for reading this paper.

2 A new datatype constructor

When referring to ML in this paper, we shall only consider a very small subset of the language, a kind of mini-ML, which contains polymorphism, pattern matching, `let`, datatypes, and recursion. We shall not consider modules, abstract datatypes, references, records, and several other features. This restriction is intended to focus our attention. It is not clear how well the extension described here will interact with the full definition of Standard ML [7] but the main ideas of integrating higher-order abstract syntax with ML can be seen within this restriction.

The first-order syntax for the untyped λ-calculus can be declared using the following ML datatype definition.

```
datatype ltm = app of ltm * ltm | abs of string * ltm | var of string;
```

The λ-term $\lambda x(fxx)$ can be encoded by the term

```
abs("x", app(app(var "f", var "x"), var "x"))
```

Before this definition can be meaningfully used, it is necessary to write several functions for testing for alphabetic variants, changing bound variable names, testing for free or bound occurrences, etc. ML itself does not treat bound variables directly. For example, while the ML term

```
abs("y", app(app(var "f", var "y"), var "y"))
```

denotes the same "abstract" λ-term, a user defined function is necessary in order to establish this fact.

ML_λ is the result of extending (mini-)ML with the following items.

1. One new type constructor: `'a => 'b` denotes the type of a λ-term with an abstracted variable of type `'a` over a term of type `'b`. We shall assume that both `'a` and `'b` are equality types. As we shall see below, `'a => 'b` will then also be an equality type. We shall furthermore restrict the type `'a` to be a user defined type. That is, it cannot be a type like `int` or `string` nor can it be a pair or list type. The reason for this restriction is that the type `'a` will be treated as an "open" type, that is, new constants of that type will appear during computations. Some concepts, such as integers and pairs, should be considered closed. This type should not be confused with the type `'a -> 'b`.

2. Two new term constructors:

   (a) `x\t : 'a => 'b` if `x` is an identifier of type `'a` and `t` is a term of type `'b`. Here, identifiers should be taken to be of the same class as the value constructor class Con of [7].

   (b) `t~x : 'b` if `t` is of type `'a => 'b` and `x` is an identifier introduced with \ and is of type `'a`. This symbol will only appear with pattern variables of `=>` type. This infix symbol can be thought of as having the type `('a => 'b) * 'a -> 'b`.

3. One new expression constructors `fun fname tok = exp1  ==> exp2`. This construction is necessary in order to extend the definitions of functions in scopes where new identifiers are introduced via \. The precedence of `==>` is higher than that of function definition.

We describe each of these extensions in turn.

Given the new type constructor above, we shall introduce three datatypes that will be used throughout the rest of this paper.

```
datatype tm = abs of tm => tm | app of tm * tm;

datatype term = a | b | f of term | g of term * term;

datatype form =  p  of term         | q    of term * term
```

```
            | and of form * form  | or   of form * form
            | imp of form * form  | not  of form
            | all of term => form | some of term => form;
```

Here, the type `tm` is the type of untyped λ-terms, while `term` and `form` are the types for first-order terms and first-order formulas, resp. Notice that in each case, where a bound variables is intended, the type constructor `=>` is used. For example, to form a universally quantified expression, the constructor `all` is applied to a ML_λ λ-term. The λ-expressions for the S, K, I combinators would be the following ML_λ terms

```
abs x\(abs y\ (abs z\(app(app(x,z),app(y,z)))))
abs x\(abs y\x)
abs x\x
```

Similarly, the first-order formula $\forall x(p(x) \supset q(f(x), a)) \wedge \forall y \exists x(q(x, g(y, x)))$ would be the following term

```
  and(all x\(imp(p(x), q(f(x),a)), all y\(some x\(q(x,g(y,x)))))
```

3 Equality and pattern matching

There would be no force to this extension if ML_λ did not have built into it some equational facts about λ-terms. In particular, these terms will satisfy the equations for α and η-conversion along with the following very weak form of β conversion

`(x\t)~x = t` (β_0).

(This equation is only required when pattern variables of `=>` type are used.) Given this kind of equality theory for ML_λ terms, it is not possible to destruct a λ-term by separating its bound variable from its body, since that operation is not invariant under α-conversion. Destructuring can be done, however, by suitably extending the notion of pattern matching.

A pattern variable, say `M` of type `t1 => t2 => ... => tn => t` (`=>` associates to the right) is permitted in a pattern/expression combination if every occurrence of `M` in that combination is of the form `M~x1~...~xm` where `m` is less than or equal to `n` and `x1,...,xm` is a list of distinct `\`-bound variables within the pattern or expression. Consider the following patterns and values for which they are to be matched.

```
(1)  x\y\(f(H~x))               u\v\(f(f(u)))
(2)  x\y\(f(H~x))               u\v\(f(f(v)))
(3)  x\y\(g(H~y~x,(f(L~x))))    u\v\(g(u,f(u)))
(4)  x\y\(g(H~x,L~x))           u\v\(g(g(a,u),g(u,u)))
```

In each of these examples, a pattern variable is written with a capital letter. Solving these patterns over the theory of α, β_o, η is a very simple generalization of first-order pattern matching. The following are the substitutions for solving these match problems.

```
(1)  H == w\(f(w))
(2)  match failure
(3)  H == y\x\x        L == x\x
(4)  H == x\(g(a,x))   L == x\(g(x,x))
```

The match failure for (2) arises from the fact that substitution for λ-terms must avoid variable capture. Hence, it is not possible to substitution a term for H in line (2) so that y is captured. This aspect of pattern matching is very useful. For example, the pattern `all x\(and(P,Q~x))` (with pattern variables P and Q) would match with a term denoting a universally quantifed conjunction in which the first conjunct does not contain a free occurrence of the quantified variable. For a more complete treatment of unification with variables of higher-type restricted as above see [10]. Unification in such a setting is like unification for first-order logic in that unification problems are decidable and most general unifiers exist when unifiers exist.

Given this use of pattern variables, the simplest functions that we can write in ML_λ would be the following:

```
fun vacuousp (x\T) = true
  | vacuousp S     = false;
(* ...   or using wild-cards  ... *)
fun vacuousp (x\_) = true
  | vacuousp _     = false;

exception DISCHARGE;
fun discharge (x\M) = M
  | discharge   _    = raise DISCHARGE
```

The function `vacuousp` has type `('a => 'b) -> bool` and can be used to determine whether or not an abstraction is vacuous. The function `discharge` has type `('a => 'b) -> 'b`. It returns the body of a vacuous abstraction or raises an exception if the abstraction is not vacuous. As the second way of writing `vacuousp` illustrates, the wildcard `_` denotes a pattern variable and that variable behaves similarly to other variables. It is not a "textual" variable; for example, the expression `x\_` does not match any λ-abstraction, it only matches a vacuous one. Use either `x\(_~x)` or simply `_` to match any λ-abstraction.

For another example, consider the following function that determines whether or not its argument is a term that denotes a Church numeral, that is, an untyped λ-term of the form $\lambda x \lambda f. f^n x$ for some $n \geq 0$.

```
fun numeralp (abs x\(abs f\x)) = true
  | numeralp (abs x\(abs f\(app(f,M~x~f)))) = numeralp (abs x\(abs f\(M~x~f)))
  | numeralp _ = false;
```

4 Extending function definitions

As the last example using Church numeral illustrates, while certain simple recursions over λ-terms is possible, more general recursions are not possible. For example, it is not possible to write a function that counts the number of applications `app` in a term of type `tm`. For more flexible recursion, we need the ability to extend datatypes, such as `tm` and `term`, with new constants and to also extend the definition of functions to include these constants. To motivate this, consider the following description of the syntax of simply typed λ-terms. Let Σ be a signature, that is, a set of simply typed constants. A term $ct_1 \cdots t_n$ (where c is neither an application nor abstraction) is a Σ-term of type τ over this signature if Σ contains the constant c at type $\tau_1 \to \cdots \to \tau_n \to \tau$ and for $i = 1, \ldots, n$, t_i is a Σ-term of type τ_i. The syntax rule for λ-abstraction is given by: $\lambda x.t$ is a Σ-term of type $\tau' \to \tau''$ if t is a $\Sigma \cup \{x : \tau'\}$-term of type τ'' (assuming x is not in Σ). That is, passing through an abstraction causes the signature (set of constants) to be increased: a bound variable can be thought of as introducing a "scoped constant."

Given this model of syntax, a functional program that performs recursion on the structure of λ-terms will need to have new constants introduced to stand for bound variables and will need to have procedures for extending functions so that they will behave correctly on those new terms. We illustrate such processing by writing one of the simplest recursive programs, the identity function. The functions `copyterm` and `copyform` defined below are such that they return their arguments unchanged by recursively copying that argument into its output.

```
fun copyterm a = a
  | copyterm b = b
  | copyterm (f X) = f(copyterm X)
  | copyterm (g(X,Y)) = g(copyterm X, copyterm Y);

fun copyform (p X)      = p(copyterm X)
  | copyform (q(X,Y))   = q(copyterm X, copyterm Y)
  | copyform (and(X,Y)) = and(copyform X, copyform Y)
  | copyform (or(X,Y))  = or(copyform X, copyform Y)
  | copyform (imp(X,Y)) = imp(copyform X, copyform Y)
  | copyform (all M)    = all  x\(fun copyterm x = x ==> copyform (M~x))
  | copyform (some M)   = some x\(fun copyterm x = x ==> copyform (M~x))
```

The only new item in these lines is in the clauses for copying `all` and `some`. Here, the `==>` expression construction is used. Evaluating the following expression

```
fun fname tok = exp1  ==> exp2
```

first extends the definition of the `fname` function with the clause that says that the \-identifier `tok` rewrites to the expression `exp1`; second, evaluates the expression `exp2`; and third, discharges the function definition extension once a value is returned.

Consider computing the value `copyform (some u\(all v\(p(u,v))))`. This would yeild the following sequence of expressions.

```
some x\(fun copyterm x = x ==> copyform (u\(all v\(p(u,v)))~x)
some x\(fun copyterm x = x ==> copyform (all v\(p(x,v))))
some x\(fun copyterm x = x ==>
  all y\(fun copyterm y = y ==> copyform (v\(p(x,v))~y)))
some x\(fun copyterm x = x ==>
  all y\(fun copyterm y = y ==> copyform (p(x,y))))
some x\(fun copyterm x = x ==>
  all y\(fun copyterm y = y ==> p(copyterm x,copyterm y)))
some x\(fun copyterm x = x ==> all y\(fun copyterm y = y ==> p(x,y)))
```

The final expression is, of course, more simply `some x\(all y\(p(x,y)))`, which is α-convertible to the initial term.

These copy-functions can be used to implement substitution into formulas. Consider the following ML_λ program.

```
fun substform M T = discharge x\(fun copyterm x = T ==> copyform (M~x))
```

Here `substform` has the type `(term => form) -> term -> form`. Its operation can be describe as follows: introduce a new identifier `x` of type `term`, instruct `copyterm` to copy `x` to `T` and then return the result of `copyform (M~x)`. Since `x` is not copied to itself, `x` cannot appear in the value `copyform (M~x)` since it is not in either the values `M` or `T`.. Thus, `discharge` will always be given a vacuous abstraction and hence will not raise an exception.

Substititution for the untyped λ-terms can be implemented similarly.

```
fun copy (app(S,T)) = app(copy S, copy T)
  | copy (abs M)    = abs(x\(fun copy x = x ==> copy (M~x)));

fun subst M T = discharge x\(fun copy x = T ==> copy (M~x));
```

Notice that in all the cases above, the coding of substitution is particularly simple and natural. The reader should consider writing similar subsitution functions in ML on the first-order syntax for untyped λ-terms and compare them to the above implementations.

In these examples, the substitution functions have been written in curried form. This choice was made simply to illustrate that substitution can be used to carry `=>` to `->`. For example, if `M` is of the type `term => form` then `substform M` is of the type `term -> form`. In otherwords, `M` can be seen as code describing a function from `term` to `form` and it is the `substform` function that translates that code into that actual function.

5 More examples

In this section we present several examples of computing on first-order formulas and on untyped λ-terms.

The following program simply counts the number of applications in an untyped λ-term.

```
fun count (app(T,S)) = 1 + count T + count S
  | count (abs M) = discharge x\(fun count x = 0 ==> count (M˜x));
```

Notice that in this function, the base cases for recursion are introduced during the execution of this function.

The following two programs perform call-by-name and call-by-value reductions on untyped λ-terms.

```
fun cbn (abs M) = abs M
  | cbn (app(T,S)) =
     let val (abs M) = cbn T in
         cbn (subst M S)
     end;

fun cbv (abs M) = abs M
  | cbv (app(T,S)) =
     let val (abs M) = cbv T in
        cbv (subst M (cbv S))
     end;
```

The following function normal computes the $\beta\eta$-normal form of an untyped λ-terms (when such normal forms exist).

```
fun onepass (app(abs M, T)) = onepass (subst M T)
  | onepass (abs x\(app(T,x))) = onepass T
  | onepass (app(T,S)) = app(onepass T, onepass S)
  | onepass (abs M) = abs x\(fun onepass x = x ==> onepass (M˜x));

fun normal T =
  let val S = onepass T in
    if T = S then T else normal S
  end;
```

The first clause of onepass reduces β-redexes and the second clause reduces η-redexes. Notice that the proviso that the abstracted varaible x in an η-reduce is not free in the term T is handled automatically.

The following program computes the negation normal form of first-order formulas by removing implications and by pushing negations using deMorgan's laws until they have atomic scopes.

```
fun nnf (p T) = p T
  | nnf (q(T,S)) = q(T,S)
  | nnf (neg(p T)) = neg(p T)
  | nnf (neg(q(T,S))) = neg(q(t,S))
  | nnf (neg(neg M)) = nnf M
  | nnf (and(M,N)) = and(nnf M, nnf N)
  | nnf ( or(M,N)) =  or(nnf M, nnf N)
```

```
  | nnf (imp(M,N)) = or(nnf(neg M),nnf N)
  | nnf (neg(imp(M,N)) = and(nnf M, nnf(neg N))
  | nnf (neg(and(M,N))) = or(nnf(neg M), nnf(neg N))
  | nnf (neg(or(M,N))) = and(nnf(neg M), nnf(neg N))
  | nnf (forall M) = forall x\(nnf(M~x))
  | nnf (exists M) = exists x\(nnf(M~x))
  | nnf (neg(forall M)) = exists x\(nnf(neg(M~x)))
  | nnf (neg(exists M)) = forall x\(nnf(neg(M~x)));
```

Notice that this function does not need to use the `==>` construction to extend a function since new constants of type `term` are handled correctly by the first four clauses.

The following functions compute a prenex normal form of a formula in negation normal form. The `merge` auxillary function is used to combine the prefixes of two formulas in prenex normal form. If the first argument to `merge` is `true`, this merging is assumed to be across a conjunction; if that argument is `false`, it is assume to be across a disjunction.

```
fun merge (true,  all M,  all N)  = all  x\(merge(true,  M~x, N~x))
  | merge (false, some M, some N) = some x\(merge(false, M~x, N~x))
  | merge (flag, (all M), N)  = all  x\(merge(flag, M~x, N))
  | merge (flag, (some M), N) = some x\(merge(flag, M~x, N))
  | merge (flag, N, (all M))  = all  x\(merge(flag, N, M~x))
  | merge (flag, N, (some M)) = some x\(merge(flag, N, M~x))
  | merge (true, M, N)  = and(M,N)
  | merge (false, M, N) =  or(M,N);

fun prenex (p T) = p T
  | prenex (q(T,S)) = q(T,S)
  | prenex (not M) = not M
  | prenex (and(P,Q)) = merge(true,  prenex P, prenex Q)
  | prenex (or(P,Q))  = merge(false, prenex P, prenex Q)
  | prenex (all M)  = all  x\(prenex(M~x))
  | prenex (some M) = some x\(prenex(M~x));
```

6 Some problems and possible variations

The design of new programming languages and extensions to old languages is a serious business that should be carefully considered. The informal presentation in this paper is certainly not such a serious study. All that is indicated here is that there might be a dimension in which ML can be extended to address the concerns of handling the data structures containing bound variables. In this section, some problems with ML_λ are mentioned and possible variations of it are considered.

6.1 Problems regarding function definition extension

When a new constant is introduced by the \ construct, it is important to know if all the necessary functions have been extended to correctly handle that new constant. In all the examples in this paper, it was an easy matter to check the calling structure of functions to be sure that suitable definition extensions were actually done. When examples get to be larger, such checks might get to be very difficult. If higher-order programming is also involved in recursions over the structure of λ-terms, then it would be impossible in general to have a static check determine that all functions that might be passed in as values are extended correctly.

Another problem with function definition extensions is that functions can only be extended to handle just the new constant: more general patterns involving that constant are not possible. For example, consider the following two programs. Both seem quite sensible as computations while they make use of a stronger form of function definition extension than is permitted above.

```
fun eq (app(T,S),app(U,V)) = eq (T,U) andalso eq (S,V)
  | eq (abs M, app N) = discharge x\(fun eq (x,x) = true ==> eq (M^x,N^x))
  | eq (_,_) = false

fun subst M t =
  let fun aux x\x = t
        | aux x\(app (M^x, N^x)) = app(aux M, aux N)
        | aux x\(abs (M^x)) = abs y\(fun aux x\y = y ==> aux x\(M^x^y))
   in aux M
  end;
```

The function `eq` of type `(tm * tm) -> bool` determines whether or not its arguments are alphabetic variants. The function definition extension `fun eq (x,x) = true` involves extending `eq` on the value `(x,x)` and not just the identifier `x`. The function `subst` reimplements the previously given function of the same type, except this time it does not use `discharge` and the `copy` function. Here again, the extension `fun aux x\y = y` is more general than permitted earlier. Notice that in both of these cases, the only constants of type `tm` that are permitted in new patterns are those involving the new constant.

6.2 Weakening pattern matching

In ML, the cost of pattern matching is dominated by the size of the pattern and not the value being matched against. This is not true of ML$_\lambda$ since pattern matching may need to check if an abstraction is vacuous in a given input and this check will require a possible descent of the entire input. It is possible to modify pattern matching so that this check for vacuous abstraction is not part of the matching process. This can be done by requiring that if a pattern variable is in the scope of \-abstracted identifiers, that pattern variable must be applied to all those abstracted variables (in some order). Thus the pattern `all x\(and(P,Q^x))` would not be permitted while the pattern `all x\(and(P^x,Q^x))` would be permitted. The check for vacuous abstraction can be programmed in this weaker language on a per-signature basis (a fully polymorphic `vacuousp` presented earlier is not possible). For example, the following program is of type `(term => term) -> bool`.

```
fun vacuoustermp x\a = true
  | vacuoustermp x\b = true
  | vacuoustermp x\(f(M~x)) = vacuoustermp M
  | vacuoustermp x\(g(M~x, N~x)) = vacuoustermp M andalso vacuoustermp N
  | vacuoustermp x\x = false;
```

There seems to be no compelling reason for making pattern matching simpler in this manner: it seems more elegant to use pattern matching to achieve the same ends as calling the various `vacuousp` functions.

6.3 Internalizing `subst`

The implementation of the various `subst` predicates in this paper was completely determined by the signature of the constants building the data structures into which substitution is done. It is therefor sensible for an implementation of ML_λ to have a generic `subst` of type `(''a => ''b) -> ''a -> ''b` for equality types `''a` and `''b`. Internalizing substitution in this way is very similar to internalizing equality in SML to certain "concrete" datatypes.

6.4 The standard litany

This preliminary report does not address the large number of questions that should be addressed in serious language design. We list and briefly comment on some of these.

1. Type inference. This should be essentially the same as type inference for ML.

2. Run time type errors. These should not be possible. The extended pattern matching process can be done without respect to type information.

3. Match exception. It is very useful to have static checks that can warn a programmer of a function definition that may not have enough cases to deal with all the constructors that can be used to build its arguments. It should be possible to extend the analysis used in ML to ML_λ, although very little can probable be done statically if higher-order programming is mixed with function definition extension.

4. Semantics of new constructions. Good question. This extension seems very intensional so getting a denotational semantics for it looks hard. On the other hand, this extension does seem to have significant "declarative context" and so should have some of meaningful semantics. Here, semantic notions used to address programs in λProlog might prove useful [12].

7 Conclusion

An extension to ML that would permit rather direct handling of data structures containing internal abstract has been informally proposed. That extension, called ML_λ, provides a way to represent such structures so that the programmer does not have direct access to bound variables names. Other more declarative ways of dealing with bound variables are made available.

References

[1] Avron, A., Honsell, F., and Mason, I (1987), *Using Typed Lambda Calculus to Implement Formal Systems on a Machine.* Technical Report ECS-LFCS-87-31, Laboratory for the Foundations of Computer Science, University of Edinburgh.

[2] Burstall, R. and Honsell, F. (1988), A natural deduction treatment of operational semantics. In *Foundations of Software Technology and Theoretical Computer Science*, pages 250–269, Springer-Verlag LNCS, Vol. 338.

[3] Coquand, T. and Huet, G. (1988), The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[4] Felty, A. and Miller, D. (1988), Specifying Theorem Provers in a Higher-Order Logic Programming Language, Proceedings of the Ninth International Conference on Automated Deduction, Argonne, IL, 23 – 26, eds. E. Lusk and R. Overbeek, Springer-Verlag Lecture Notes in Computer Science, Vol. 310, 61 – 80.

[5] Hannan, J. and Miller, D. (1988), Uses of Higher-Order Unification for Implementing Program Transformers, Fifth International Conference and Symposium on Logic Programming, ed. K. Bowen and R. Kowalski, MIT Press, 942 – 959.

[6] Hannan, J. and Miller, D. (1989), A Meta Language for Functional Programs, Chapter 24 of *Meta-Programming in Logic Programming*, eds. H. Rogers and H. Abramson, MIT Press, 453–476.

[7] Harper, R., Milner, R., and Tofte, M. (1989), *The Definition of Standard ML: Version 3.* Technical Report ECS-LFCS-89-81, Laboratory for the Foundations of Computer Science, University of Edinburgh.

[8] Harper, R., Honsell, F., and Plotkin, G. (1987), A framework for defining logics. In *Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY.

[9] Huet, G. and Lang, B. (1978), Proving and Applying Program Transformations Expressed with Second-Order Logic, *Acta Informatica* 11, 31 – 55.

[10] Miller, D. (1990), "A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification," in *Extensions of Logic Programming* edited by Peter Schroeder-Heister, Springer-Verlag.

[11] Miller, D. and Nadathur, G. (1987), A Logic Programming Approach to Manipulating Formulas and Programs, Proceedings of the IEEE Fourth Symposium on Logic Programming, IEEE Press, 379 – 388.

[12] Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. (1988), Uniform proofs as a foundation for logic programming. To appear in the Annals of Pure and Applied Logic.

[13] Nadathur, G. and Miller, D. (1988), An Overview of λProlog, Fifth International Conference on Logic Programming, eds. R. Kowlaski and K. Bowen, MIT Press, 810 – 827.

[14] Paulson, L. (1986), Natural Deduction as Higher-Order Resolution, Journal of Logic Programming **3**, 237 – 258.

[15] Pauslon, L. (1989), The Foundation of a Generic Theorem Prover, Journal of Automated Reasoning, Vol. 5, 363 – 397.

[16] Pfenning, F. and Elliott, C. (1988), Higher-Order Abstract Syntax, Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, ACM Press, 199 – 208.

[17] Pfenning, F. (1989), Elf: A Language for Logic Definition and Verified Metaprogramming, Fourth Annual Symposium on Logic in Computer Science, Monterey, CA, 313 – 321.

A series of type theories and their interpretations in the logical theory of constructions

Nax Paul Mendler

University of Manchester

Abstract

We present three Martin-Löf type theories in a natural deduction style: TT_0, a type theory without a type universe; TT_1, a type theory with one type universe; and TT_ω, a type theory with a cumulative hierarchy of type universes. Then we give interpretations of each in a corresponding LTC theory.

1 Introduction

This is a companion article to "The Logical Theory of Constructions: a formal framework and its implementation" [1], where three languages in the logical theory of constructions [6] were presented: LTC_0, LTC_1 and LTC_ω. Here we present three Martin-Löf-style type theories [7, 1, 7]: TT_0, a type theory without type universes; TT_1, a type theory with one type universe; and TT_ω, a type theory with a cumulative hierarchy of type universes. (I'm assuming the reader is familiar with such type theories. See [1] or [7] for an introduction to Martin-Löf type theory.)

Then we give interpretations of each in a corresponding LTC theory. The style of the proof rules for the type theories was chosen so that the interpretation of the rules could be easily expressed in a theorem prover like Isabelle [8], which is suited to implementing object logics expressed in a natural deduction style. In such a setting one can then formally verify the soundness of the interpretation.

In type theories, equality is problematic. Here we are not thinking of equality as definitional, as it is sometimes done. Instead, equality between types is taken to be extensional, and the equality on a type's members has a similar extensional flavor: for instance, in a Π type, terms representing functions are equal if they map equal terms in the domain to equal terms in the codomain.

2 The type theories TT_0, TT_1 and TT_ω

The type theories we present here are expressed in a metalanguage with three atomic sorts: a sort of terms (o), a sort of type expressions (τ) and a sort of judgments (jud). We allow higher-order sorts to be formed with $\rightarrow$, for example: $o \rightarrow \tau$, $(o \rightarrow o) \rightarrow o$ and so on. We can build expressions of higher-order sorts in the usual fashion: for example, if E is an expression of sort $\tau \rightarrow \tau \rightarrow jud$, then $(A : \tau)E(A)(A)$ is of sort $\tau \rightarrow jud$. $E(A, A)$, for example, is another notation for $E(A)(A)$.

There are two forms of judgment: type equality, and term equality in a type:

$$\_ = \_ : \tau \rightarrow \tau \rightarrow jud \tag{11}$$

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

$$\_ = \_ \in \_ : o \to o \to \tau \to jud \tag{12}$$

For convenience, we define notations for their reflexive instances:

$$A\ Type \equiv A = A \tag{13}$$
$$a \in A \equiv a = a \in A \tag{14}$$

This is just a notational abbreviation: only the first two forms of judgment are basic.

The proof rules for each type theory will be given in a natural deduction style, where the basic unit of assertion will be a judgment.

2.1 Type theory TT_0

Our first type theory is a type theory without a type universe. TT_0 has the following list of term constructors.

$$\lambda : (o \to o) \to o \tag{15}$$
$$Ap : o \to o \to o \tag{16}$$
$$\langle \_, \_ \rangle : o \to o \to o \tag{17}$$
$$Spread : o \to (o \to o \to o) \to o \tag{18}$$
$$inl : o \to o \tag{19}$$
$$inr : o \to o \tag{20}$$
$$Decide : o \to (o \to o) \to (o \to o) \to o \tag{21}$$
$$any : o \to o \tag{22}$$
$$true : o \tag{23}$$
$$0 : o \tag{24}$$
$$S : o \to o \tag{25}$$
$$Ind : o \to o \to (o \to o \to o) \to o \tag{26}$$
$$peano : o \tag{27}$$

TT_0 has the following list of type constructors.

$$\Pi : \tau \to (o \to \tau) \to \tau \tag{28}$$
$$\Sigma : \tau \to (o \to \tau) \to \tau \tag{29}$$
$$\_ + \_ : \tau \to \tau \to \tau \tag{30}$$
$$I : \tau \to o \to o \to \tau \tag{31}$$
$$Void : \tau \tag{32}$$
$$Nat : \tau \tag{33}$$

The logic for deriving judgments is given in a natural deduction style. In order to make the interpretation into the LTC simpler (and directly expressible in Isabelle) these rules have the feature that they always bind new variables in pairs. For example, the rule for forming equal Π types is:

$$\frac{A = A' \quad \begin{array}{c} x = x' \in A \\ \vdots \\ B(x) = B'(x') \end{array}}{\Pi(A, B) = \Pi(A', B')}$$

where x and x' are assumed to be new variables whose instances are bound in the right-hand subproof, and the hypothesis $x = x' \in A$ is discharged, as usual, by this proof rule. So in the Isabelle presentation, this rule would be written:

$$(A = A') \Rightarrow (\bigwedge_{x,x'} (x = x' \in A) \Rightarrow (B(x) = B'(x'))) \Rightarrow (\Pi(A, B) = \Pi(A', B'))$$

(As usual, we are implicitly quantifying over $A, A' : \tau$ and $B, B' : o \to \tau$ in this Isabelle expression.) The proof rules for TT_0 are the following.

General rules

$$\frac{a = a' \in A, \quad A\ \mathit{Type}}{a' = a \in A} \qquad \frac{A = A'}{A' = A}$$

$$\frac{a = a' \in A, \quad a' = a'' \in A, \quad A\ \mathit{Type}}{a = a'' \in A} \qquad \frac{A = A', \quad A' = A''}{A = A''}$$

$$\frac{a = a' \in A, \quad A = A'}{a = a' \in A'}$$

$$\frac{\begin{array}{c} x = x' \in A \\ \vdots \\ B(x) = B'(x') \end{array} \quad a = a' \in A}{B(a) = B'(a')} \qquad \frac{\begin{array}{c} x = x' \in A \\ \vdots \\ b(x) = b'(x') \in B(x) \end{array} \quad a = a' \in A}{b(a) = b'(a') \in B(a)}$$

Π rules

$$\frac{A = A' \quad \begin{array}{c} x = x' \in A \\ \vdots \\ B(x) = B'(x') \end{array}}{\Pi(A,B) = \Pi(A',B')} \qquad \frac{\begin{array}{c} x = x' \in A \\ \vdots \\ f(x) = f'(x') \in B(x) \end{array}}{\lambda(f) = \lambda(f') \in \Pi(A,B)}$$

$$\frac{c = c' \in \Pi(A,B) \quad a = a' \in A}{Ap(c,a) = Ap(c',a') \in B(a)} \frac{c = c' \in \Pi(A,B)}{c = \lambda((x:o)Ap(c',x)) \in \Pi(A,B)}$$

Σ rules

$$\frac{A = A' \quad \begin{array}{c} x = x' \in A \\ \vdots \\ B(x) = B'(x') \end{array}}{\Sigma(A,B) = \Sigma(A',B')} \qquad \frac{a = a' \in A, \quad b = b' \in B(a)}{\langle a,b\rangle = \langle a',b'\rangle \in \Sigma(A,B)}$$

$$\frac{c = c' \in \Sigma(A,B) \quad \begin{array}{c} x = x' \in A,\ y = y' \in B(x) \\ \vdots \\ f(x,y) = f'(x',y') \in C(\langle x,y\rangle) \end{array} \quad \begin{array}{c} z = z' \in \Sigma(A,B) \\ \vdots \\ C(z) = C(z') \end{array}}{Spread(c,f) = Spread(c',f') \in C(c)}$$

$+$ rules

$$\frac{A = A', \quad B = B'}{A + B = A' + B'} \qquad \frac{a = a' \in A}{inl(a) = inl(a') \in A + B} \qquad \frac{b = b' \in B}{inr(b) = inr(b') \in A + B}$$

$$\frac{c = c' \in A + B \quad \begin{array}{c} x = x' \in A \\ \vdots \\ f(x) = f'(x') \in C(inl(x)) \end{array} \quad \begin{array}{c} x = x' \in B \\ \vdots \\ g(x) = g'(x') \in C(inr(x)) \end{array} \quad \begin{array}{c} z = z' \in A + B \\ \vdots \\ C(z) = C(z') \end{array}}{Decide(c,f,g) = Decide(c',f',g') \in C(c)}$$

Void **rules**

$$\frac{}{Void\ Type} \qquad \frac{a = a' \in Void}{any(a) = any(a') \in A(a)}$$

I **rules**

$$\frac{a = a' \in A, \quad b = b' \in A, \quad A = A'}{I(A, a, b) = I(A', a', b')} \frac{a = b \in A}{true = true \in I(A, a, b)}$$

$$\frac{c \in I(A, a, b), \quad c' \in I(A, a, b)}{c = c' \in I(A, a, b)} \qquad \frac{c = c' \in I(A, a, b)}{a = b \in A}$$

Nat **rules**

$$\frac{}{Nat\ Type} \qquad \frac{}{0 \in Nat} \qquad \frac{a = a' \in Nat}{S(a) = S(a') \in Nat}$$

$$\frac{a = a' \in Nat \quad \begin{array}{c} x = x' \in Nat \\ \vdots \\ A(x) = A(x') \end{array} \quad b = b' \in A(0) \quad \begin{array}{c} x = x' \in Nat,\ y = y' \in A(x) \\ \vdots \\ f(x, y) = f'(x', y') \in A(S(x)) \end{array}}{Ind(a, b, f) = Ind(a', b', f') \in A(a)}$$

$$\frac{0 = S(0) \in Nat}{peano \in Void}$$

The last rule is necessary in a type theory without a universe type, because such theories admit trivial models where all terms of a given type are equal. But even in a type theory with no universe types, we want to be able to prove zero is not equal to one!

Computation rules

All the computation rules are gathered together here because they are all of the form: redex equals contractum in type if either one of them is in the type. These rules are easily shown. We have been influenced by the NUPRL [1] style of "direct computation" rules, which exploit the fact that the interpretation is by untyped terms. The more usual computational rules are also sound in our interpretations, so they could be substituted for these stronger rules.

$$\frac{f(a) \in D}{Ap(\lambda(f), a) = f(a) \in D} \qquad \frac{Ap(\lambda(f), a) \in D}{Ap(\lambda(f), a) = f(a) \in D}$$

$$\frac{f(a, b) \in D}{Spread(\langle a, b\rangle, f) = f(a, b) \in D} \qquad \frac{Spread(\langle a, b\rangle, f) \in D}{Spread(\langle a, b\rangle, f) = f(a, b) \in D}$$

$$\frac{f(a) \in D}{Decide(inl(a), f, g) = f(a) \in D} \qquad \frac{Decide(inl(a), f, g) \in D}{Decide(inl(a), f, g) = f(a) \in D}$$

$$\frac{g(b) \in D}{Decide(inr(b), f, g) = g(b) \in D} \qquad \frac{Decide(inr(b), f, g) \in D}{Decide(inr(b), f, g) = g(b) \in D}$$

$$\frac{b \in D}{Ind(0, b, f) = b \in D} \qquad \frac{Ind(0, b, f) \in D}{Ind(0, b, f) = b \in D}$$

$$\frac{f(a, Ind(a, b, f)) \in D}{Ind(S(a), b, f) = f(a, Ind(a, b, f)) \in D} \qquad \frac{Ind(S(a), b, f) \in D}{Ind(S(a), b, f) = f(a, Ind(a, b, f)) \in D}$$

2.2 Type theory TT_1

We extend type theory TT_0 by adding a universe, a type whose elements are indices of other types, and which are mapped to types by the new constructor Ty. The term constructors of TT_1 are the term constructors of TT_0 (15–27) plus the list:

$$\dot{\Pi}{:}o \to (o \to o) \to o \tag{34}$$
$$\dot{\Sigma}{:}o \to (o \to o) \to o \tag{35}$$
$$\_\dot{+}\_{:}o \to o \to o \tag{36}$$
$$\dot{I}{:}o \to o \to o \to o \tag{37}$$
$$\dot{Void}{:}o \tag{38}$$
$$\dot{Nat}{:}o \tag{39}$$

The type constructors of TT_1 are the type constructors of TT_0 (28–33) plus:

$$U_1{:}\tau \tag{40}$$
$$Ty{:}o \to \tau \tag{41}$$

The proof rules of TT_1 are the proof rules of TT_0 plus the following.

U_1 **rules**

$$\frac{}{U_1 \ \mathit{Type}} \qquad \frac{a = a' \in U_1}{\mathit{Ty}(a) = \mathit{Ty}(a')}$$

$$\frac{}{\dot{\mathit{Void}} \in U_1} \qquad \frac{}{\mathit{Ty}(\dot{\mathit{Void}}) = \mathit{Void}}$$

$$\frac{}{\dot{\mathit{Nat}} \in U_1} \qquad \frac{}{\mathit{Ty}(\dot{\mathit{Nat}}) = \mathit{Nat}}$$

$$\frac{a = a' \in U_1, \quad b = b' \in U_1}{a \dot{+} b = a' \dot{+} b' \in U_1} \qquad \frac{\mathit{Ty}(a \dot{+} b) \ \mathit{Type}}{\mathit{Ty}(a \dot{+} b) = \mathit{Ty}(a) + \mathit{Ty}(b)}$$

$$\frac{a = a' \in U_1, \quad b = b' \in \mathit{Ty}(a), \quad c = c' \in \mathit{Ty}(a)}{\dot{I}(a, b, c) = \dot{I}(a', b', c') \in U_1} \qquad \frac{\mathit{Ty}(\dot{I}(a, b, c)) \ \mathit{Type}}{\mathit{Ty}(\dot{I}(a, b, c)) = I(\mathit{Ty}(a), b, c)}$$

$$\frac{a = a' \in U_1 \quad \begin{matrix} x = x' \in \mathit{Ty}(a) \\ \vdots \\ b(x) = b'(x') \in U_1 \end{matrix}}{\dot{\Pi}(a, b) = \dot{\Pi}(a', b') \in U_1} \qquad \frac{\mathit{Ty}(\dot{\Pi}(a, b)) \ \mathit{Type}}{\mathit{Ty}(\dot{\Pi}(a, b)) = \Pi(\mathit{Ty}(a), \mathit{Ty} \circ b)}$$

$$\frac{a = a' \in U_1 \quad \begin{matrix} x = x' \in \mathit{Ty}(a) \\ \vdots \\ b(x) = b'(x') \in U_1 \end{matrix}}{\dot{\Sigma}(a, b) = \dot{\Sigma}(a', b') \in U_1} \qquad \frac{\mathit{Ty}(\dot{\Sigma}(a, b)) \ \mathit{Type}}{\mathit{Ty}(\dot{\Sigma}(a, b)) = \Sigma(\mathit{Ty}(a), \mathit{Ty} \circ b)}$$

2.3 Type theory TT_ω

Our final type theory has a cumulative hierarchy of universes. Let i, j vary over the numerals 1,2,3... The term constructors of TT_ω are the term constructors of TT_1 (15–27, 34–39) plus:

$$\dot{U}_i{:}o \tag{42}$$

for $i = 1, 2, 3, \ldots$ The type constructors of TT_ω are the type constructors of TT_1 (28–33,40–41) plus:

$$U_i{:}\tau \tag{43}$$

for $i = 2, 3, \ldots$ (no need to define U_1 twice!). The proof rules of TT_ω are the proof rules of TT_1, now with the universe rules given for every level i , plus rules asserting that $\dot{U}_i$ is the reflection of U_i and that the universes are cumulative:

Additional U_i rules

$$\frac{}{\dot{U}_i \in U_j}\ i < j \qquad \frac{}{Ty(\dot{U}_i) = U_i} \qquad \frac{a = a' \in U_i}{a = a' \in U_{i+1}}$$

3 Interpretation of the type theories in the LTC

We now show how to interpret each type theory in the corresponding LTC language. These interpretations are similar to the interpretations given by Jan Smith in [9, 10], and Stuart Allen in [2, 3]. In fact, our pattern of binding new variables in pairs in the proof rules gives an encoding of their notions of "true sequent."

For the language of the LTC, let *term* be the sort of terms and *form* be the sort of formulas. In all three interpretations we interpret the different syntactic classes as follows.

- Expressions of sort o (term expressions) will be interpreted as LTC terms.
- Expressions of sort τ (type expressions) will be interpreted as binary predicates.
- Expressions of sort *jud* (judgments) will be interpreted as formulas.

So we have the interpretations:

$$[\![o]\!] \equiv term \tag{44}$$
$$[\![\tau]\!] \equiv term \to term \to form \tag{45}$$
$$[\![jud]\!] \equiv form \tag{46}$$

This induces an interpretation of the type theory rules as rules of the LTC, and one can show these are all *derived* rules, verifying the soundness of the interpretation.

In each interpretation, the judgments will be interpreted as follows. The judgment $A = A'$ will be interpreted as the formula asserting that the interpretations of A and A' are logically equivalent and are also partial equivalence relations on canonical values — that is, they are symmetric, transitive and relate only *canonical values* (terms that evaluate to themselves) — the evaluation relation of the LTC is written "$a \rightsquigarrow b$". Thus:

$$\begin{aligned}[\![A = A']\!] \equiv\, & (\forall u, u'.\ [\![A]\!](u, u') \Leftrightarrow [\![A']\!](u, u')) \wedge \\ & (\forall u, u'.\ [\![A]\!](u, u') \Rightarrow [\![A]\!](u', u)) \wedge \\ & (\forall u, u', u''.\ [\![A]\!](u, u') \Rightarrow [\![A]\!](u', u'') \Rightarrow [\![A]\!](u, u'')) \wedge \\ & \forall u.\ [\![A]\!](u, u) \Rightarrow u \rightsquigarrow u\end{aligned} \tag{47}$$

The judgment $a = a' \in A$ will be interpreted as the formula asserting that the interpretations of a and a' evaluate to values related by the interpretation of A. Thus:

$$[\![a = a' \in A]\!] \quad \equiv \quad [\![a]\!] = [\![a']\!]\ in\ [\![A]\!] \tag{48}$$

where, for convenience, we introduce the notational abbreviation $\_ = \_ \ in \ \_$ (of sort $term^2 \rightarrow (term^2 \rightarrow form) \rightarrow form$), defined as:

$$a = a' \ in \ A \quad \equiv \quad \exists u, u'.a \rightsquigarrow u \wedge a' \rightsquigarrow u' \wedge A(u, u') \tag{49}$$

It's also convenient to extend the interpretation of types as binary predicates to an interpretation of families of types (sort $o \rightarrow \tau$) as ternary predicates. We define, for $B : o \rightarrow \tau$,

$$[\![B]\!] \quad \equiv \quad (x : term)[\![B(x)]\!] \tag{50}$$

where we are "punning" on the variable x: in expression $B(x)$ (of sort τ), x is taken to be a corresponding variable of sort o which will be interpreted back to x, the original variable of sort *term*. Now we give the interpretation for each type theory in turn.

3.1 The interpretation of TT_0

The interpretation of type theory terms is easily done. All the constructors except *true*, *any* and *peano* already appear as term constructors in LTC_0, so interpret them by their namesakes. And for the remaining three, we define:

$$[\![true]\!] \equiv 0 \tag{51}$$

$$[\![any]\!] \equiv (x)x \tag{52}$$

$$[\![peano]\!] \equiv (x)x \tag{53}$$

(One can check that *true* needs only to be a canonical term, and that since *any* and *peano* arise in only logically absurd situations, no properties are required of them.)

The following are the definitions of the type constructors. Note that we are using the $\exists$ quantifier over $term \rightarrow term$ in the interpretation of Π.

$$[\![\Pi(A, B)]\!](z, z') \equiv \exists f, f'. \ z = \lambda(f) \wedge z = \lambda(f') \wedge \forall x, x'.x = x' \ in \ [\![A]\!] \Rightarrow f(x) = f'(x') \ in \ [\![B]\!](x) \tag{54}$$

$$[\![\Sigma(A, B)]\!](z, z') \equiv \exists a, a', b, b'. \ z = \langle a, b \rangle \wedge z' = \langle a', b' \rangle \wedge a = a' \ in \ [\![A]\!] \wedge b = b' \ in \ [\![B]\!](a) \tag{55}$$

$$[\![A + A']\!](z, z') \equiv \exists a, a'. \ z = inl(a) \wedge z' = inl(a') \wedge a = a' \ in \ [\![A]\!] \vee z = inr(a) \wedge z' = inr(a') \wedge a = a' \ in \ [\![A']\!] \tag{56}$$

$$[\![I(A, a, b)]\!](z, z') \equiv z = [\![true]\!] \wedge z' = [\![true]\!] \wedge [\![a]\!] = [\![b]\!] \ in \ [\![A]\!] \tag{57}$$

$$[\![Void]\!](z, z') \equiv \perp \tag{58}$$

$$[\![Nat]\!](z, z') \equiv z \rightsquigarrow z \wedge z' \rightsquigarrow z' \wedge \exists u. \ z \leadsto u \wedge z' \leadsto u \tag{59}$$

One can show that the interpretations of the proof rules of TT_0 are derived rules of LTC_0.

3.2 The interpretation of TT_1

We extend the previous interpretation to give an interpretation of TT_1 in LTC_1. In particular, we use the reflected notion of proposition and truth present in LTC_1 to interpret the universe type U_1.

In the same way as we extended the interpretation of types to families, it is handy to extend the interpretation of terms to families of terms (sort $o \to o$) by defining, for $b : o \to o$,

$$[\![b]\!] \quad \equiv \quad (x : term)[\![b(x)]\!] \tag{60}$$

We interpret $Ty(a)$ as the binary predicate for which $[\![a]\!]$ is an internal representation.

$$[\![\, Ty(a)]\!](u, u') \equiv T([\![a]\!]uu') \tag{61}$$

We interpret U_1 as a binary predicate that holds between two canonical terms when they are both internal, binary propositional functions which are the internal representations of equal types.

$$[\![U_1]\!](z, z') \equiv z \rightsquigarrow z \wedge z' \rightsquigarrow z' \wedge (\forall u, u'.P_1(zuu') \wedge P_1(z'uu')) \wedge [\![\, Ty(z) = Ty(z')]\!] \tag{62}$$

For convenience, we define the notational abbreviation $\_ = \_ \dot{\in} \_$ of sort $term^3 \to term$ by

$$a = a' \,\dot{\in}\, b \equiv \dot{\exists} u, u'.a \,\dot{\rightsquigarrow}\, u \,\dot{\wedge}\, a' \,\dot{\rightsquigarrow}\, u' \,\dot{\wedge}\, buu' \tag{63}$$

so that $T(a = a' \,\dot{\in}\, [\![b]\!]) \Leftrightarrow a = a' \; in \; [\![\, Ty(b)]\!]$. We interpret the new type index constructors (34–39) as the internal versions of the interpretations given to the corresponding type constructors.

$$\begin{aligned}
[\![\, \dot{\Pi}(a, b)]\!] \equiv & \lambda z, z'. \dot{\exists} f, f'.z \doteq \lambda(f) \,\dot{\wedge}\, z' \doteq \lambda(f') \,\dot{\wedge} && (64)\\
& \dot{\forall} x, x'.x = x' \,\dot{\in}\, [\![a]\!] \,\dot{\supset}\, f(x) = f(x') \,\dot{\in}\, [\![b]\!](x) && \\
[\![\, \dot{\Sigma}(a, b)]\!] \equiv & \lambda z, z'. \dot{\exists} u, u', v, v''.z \doteq \langle u, v\rangle \,\dot{\wedge}\, z' \doteq \langle u', v'\rangle \,\dot{\wedge} && (65)\\
& u = u' \,\dot{\in}\, [\![a]\!] \,\dot{\wedge}\, v = v' \,\dot{\in}\, [\![b]\!](u) && \\
[\![a \dotplus a']\!] \equiv & \lambda z, z'. \dot{\exists} u, u'.(z \doteq inl(u) \,\dot{\wedge}\, z' \doteq inl(u') \,\dot{\wedge}\, u = u' \,\dot{\in}\, [\![a]\!]) \,\dot{\vee} && (66)\\
& (z \doteq inr(u) \,\dot{\wedge}\, z' \doteq inr(u') \,\dot{\wedge}\, u = u' \,\dot{\in}\, [\![a']\!]) && \\
[\![\dot{I}(a, c, c')]\!] \equiv & \lambda z, z'.z \doteq [\![true]\!] \,\dot{\wedge}\, z' \doteq [\![true]\!] \,\dot{\wedge}\, [\![c]\!] = [\![c']\!] \,\dot{\in}\, [\![a]\!] && (67)\\
[\![\dot{Void}]\!] \equiv & \lambda z, z'.\dot{\bot} && (68)\\
[\![\dot{Nat}]\!] \equiv & \lambda z, z'.z \,\dot{\rightsquigarrow}\, z \,\dot{\wedge}\, z' \,\dot{\rightsquigarrow}\, z' \,\dot{\wedge}\, \dot{\exists} u.z \,\dot{\leadsto}\, u \,\dot{\wedge}\, z' \,\dot{\leadsto}\, u && (69)
\end{aligned}$$

One can show that the interpretation of the proof rules of TT_1 are derived rules of LTC_1.

3.3 The interpretation of TT_ω

Finally, we extend the previous interpretation to give an interpretation of TT_ω in LTC_ω. First, we interpret U_i in the manner of (62):

$$[\![U_i]\!](z, z') \equiv z \rightsquigarrow z \wedge z' \rightsquigarrow z' \wedge (\forall u, u'.P_i(zuu') \wedge P_i(z'uu')) \wedge [\![\, Ty(z) = Ty(z')]\!] \tag{70}$$

Then we interpret the type index $\dot{U}_i$ as the reflection of this:

$$[\![\dot{U}_i]\!] \equiv \lambda z, z'.z \,\dot{\rightsquigarrow}\, z \,\dot{\wedge}\, z' \,\dot{\rightsquigarrow}\, z' \,\dot{\wedge}\, (\dot{\forall} u, u'.\dot{P}_i(zuu') \,\dot{\wedge}\, \dot{P}_i(z'uu')) \,\dot{\wedge}_c\, (z \doteq_{Ty} z') \tag{71}$$

where we define the notational abbreviations "conditional and" and "equal type indices" (both of sort $term^2 \to term$) as follows.

$$a \,\dot{\wedge}_c\, a' \equiv a \,\dot{\wedge}\, (a \,\dot{\supset}\, a') \tag{72}$$

$$\begin{aligned} z \doteq_{Ty} z' \equiv & (\dot{\forall} u, u'. zuu' \,\dot{\Leftrightarrow}\, z'uu') \,\dot{\wedge}\, (\dot{\forall} u, u'. zuu' \,\dot{\supset}\, zu'u) \,\dot{\wedge} \\ & (\dot{\forall} u, u', u''.\, zuu' \,\dot{\supset}\, zu'u'' \,\dot{\supset}\, zuu'') \wedge \dot{\forall} u.\, zuu \,\dot{\supset}\, u \dot{\leadsto} u \end{aligned} \tag{73}$$

Thus given $\forall u, u'.\, P_i(zuu') \wedge P_i(z'uu')$, we have $T(z \doteq_{Ty} z') \Leftrightarrow [\![Ty(z) = Ty(z')]\!]$.

As before, one can show that the interpretation of the proof rules of TT_ω are derived rules of LTC_ω.

4 Conclusion

We have presented a series of type theories, parallel to the series of LTC languages presented in [1]. Connecting them are interpretations of each type theory in its corresponding logical theory. The type theories were designed to make this interpretation easily expressible in a theorem prover like Isabelle. We have implemented LTC_ω and TT_ω in Isabelle and intend to formally prove the soundness of the interpretation, as well as developing other theories within TT_ω and LTC_ω.

References

[1] Peter Aczel and David Carlisle. The Theory of Constructions: A formal framework and its implementation. manuscript, 1990.

[2] S. F. Allen. A non-type-theoretic definition of Martin-Löf's types. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1986.

[3] S.F. Allen. *A Non-type-theoretic semantics for a type-theoretic language.* PhD thesis, Cornell University, 1987.

[4] R.L. Constable et al. *Implementing Mathematics with the* NuPrl *Proof Development System.* Prentice-Hall, 1986.

[5] P. Martin-Löf. Intuitionistic type theory. Notes by G. Sambin of a series of lectures given in Padua, June 1980, Bibliopolis, Napoli, 1984.

[6] N.P. Mendler and P. Aczel. The notion of a framework and a framework for the LTC. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, pages 392–399. IEEE, 1988.

[7] B. Nordstrom, K. Peterson, and J. Smith. *An Introduction to Martin-Löf's Theory of Types.* Oxford University Press, 1990.

[8] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, 1990.

[9] Jan Smith. *On the relation between a type theoretic and a logical formulation of the theory of constructions*. PhD thesis, Gothenberg University, 1978.

[10] Jan Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *Journal of Symbolic Logic*, 49:730–753, 1984.

Quotient types via coequalizers in Martin-Löf type theory

Nax Paul Mendler

University of Manchester

Abstract

This note describes the notion of quotient type as it arises from coequalizers in a category of types, as an exercise in working from a categorical property to arrive at its expression in the type theory. As examples, we use quotient types to internalize the notion of being an epic morphism, and to define "squash" types, and with squash types, a weak version of the subset type constructor. Some familiarity with the semantics of Martin-Löf type theory is assumed, but we'll begin with an overview. (I'm assuming the reader is familiar with such type theories. See [7], [1] or [8] for an introduction to Martin-Löf type theory.)

1 An overview of semantics

The basic paradigm in the semantics of type theory is to associate a category with a class of types: the objects are types and a morphism $t : A \to B$ is a term of type B parameterized by a type A (informally, having a free variable of type A). Type equality can be problematic and I won't be addressing the issue here. Throughout this note, category $\mathcal{C}$ is intended to be a category of types, and as we go along we will assume more structure of it, to explain the different type constructors as they are introduced.

1.1 Slice categories

In [10], Seely's key step in modeling Martin-Löf type theory was to equate families of types over a type A with morphisms having codomain A. This is a well-known technique in category theory; such a morphism can be viewed as a "generalized object (over A)". (Families of types can be modeled more generally [3], but here we take the simpliest approach.) Intuitively, we can think of a morphism $f : B \to A$ as being in the form of a projection $\coprod_{x\in A} B(x) \to A$ where $B(x) \equiv \{y \in B | f(y) = x\}$.

A morphism from family $f : B \to A$ to family $f' : B' \to A$ ought to be a family of morphisms $\{t(x)\}_{x\in A}$, with $t(x) : B(x) \to B'(x)$. This follows if $t : B \to B'$ is a morphism in $\mathcal{C}$ such that $f' \circ t = f$. Thus we are lead to consider slice categories:

Definition 1 *For any category $\mathcal{C}$ and $A \in \mathcal{C}$, the* slice category *$\mathcal{C}/A$ has, as objects, the morphisms of $\mathcal{C}$ of the form $f : B \to A$, and a morphism from $f : B \to A$ to $f' : B' \to A$ is a $t : B \to B'$ in $\mathcal{C}$ such that $f' \circ t = f$.*

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

If $\mathcal{C}$ is a category of types then we are taking $\mathcal{C}/A$ to be the category of types in a parameter type A. We introduce the following notations for slice categories, and follow with some comments.

$$\begin{array}{c} A \vdash B\ \mathit{Type} \equiv B \in \mathcal{C}/A \\ A \vdash t : B \to B' \equiv t : B \to B' \text{ in } \mathcal{C}/A \\ A \vdash t = t' : B \to B' \equiv t = t' : B \to B' \text{ in } \mathcal{C}/A \end{array} \tag{74}$$

1. The notations should be subscripted by "$\mathcal{C}$," but the base category is always clear from the context.
2. $\mathcal{C}/A$ has a terminal object, namely id_A; we introduce the notation $A \vdash t : B$ for $A \vdash t : 1 \to B$. Another abbreviation is to write $A \vdash t, t' : B \to B'$ when $A \vdash t : B \to B'$ and $A \vdash t' : B \to B'$.
3. By writing $A \vdash \mathit{nota} \equiv t : B \to C$, we introduce the notation "*nota*" for morphism t.
4. We will always have a terminal object 1 in $\mathcal{C}$, in which case $\mathcal{C}/1 \cong \mathcal{C}$. Often, these categories will be identified, thus $\vdash A\ \mathit{Type}$ and $\vdash a : A' \to A$ are shorthand for $1 \vdash A\ \mathit{Type}$ and $1 \vdash a : A' \to A$, or are statements about $\mathcal{C}$.
5. An important fact about slice categories is that if $A \vdash B\ \mathit{Type}$ then $(\mathcal{C}/A)/B \cong \mathcal{C}/\mathit{dom}(B)$, so $B \vdash C\ \mathit{Type}$, for example, can be thought of as shorthand for $\mathit{dom}(B) \vdash C\ \mathit{Type}$ (or $\Sigma(A, B) \vdash C\ \mathit{Type}$, as we shall see), or as the statement $C \in (\mathcal{C}/A)/B$.
6. These sequents can be read as a "combinator-style" type theory, but one can introduce variables and prove the expected functional completeness result (although I haven't seen anyone carry out the details – citation?).

1.2 The unit type

If $\mathcal{C}$ has a final object 1, then it models the "one element" unit type. There is a unique morphism from any other type to 1, and so we have the following rules.

$$\frac{}{\vdash 1\ \mathit{Type}} \qquad \frac{\vdash X\ \mathit{Type}}{\vdash !_X : X \to 1} \qquad \frac{\vdash t : X \to 1}{\vdash t = !_X : X \to 1}$$

We will write $!_X$ or just X for the unique morphism in $X \to 1$.

1.3 Substitution

In this setting substitution is a semantic, not just a syntactic, operation. To express it in the category, we assume $\mathcal{C}$ has pullbacks. If $A \vdash B\ \mathit{Type}$ and $\vdash a : A' \to A$, then substituting a into B yields a family of types over A':

$$\frac{A \vdash B\ \mathit{Type} \quad \vdash a : A' \to A}{A' \vdash B[a]\ \mathit{Type}} \qquad [\text{informally} : \ \frac{x : A \vdash B(x)\ \mathit{Type} \quad \vdash a : A' \to A}{y : A' \vdash B(a(y))\ \mathit{Type}}] \tag{75}$$

We construct $B[a]$ by means of the following pullback.

$$\begin{tikzcd} \bullet \arrow[r] \arrow[d, "{B[a]}"'] & \bullet \arrow[d, "B"] \\ A' \arrow[r, "a"'] & A \end{tikzcd} \tag{76}$$

If $A \vdash t : B \to B'$ then we can form the pullback squares

$$\begin{tikzcd} \bullet \arrow[r, "\epsilon"] \arrow[d, "{B[a]}"'] & \bullet \arrow[d, "B"] \\ A' \arrow[r, "a"'] & A \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} \bullet \arrow[r, "\epsilon'"] \arrow[d, "{B'[a]}"'] & \bullet \arrow[d, "B'"] \\ A' \arrow[r, "a"'] & A \end{tikzcd} \tag{77}$$

and then $A' \xleftarrow{B[a]} \bullet \xrightarrow{t \circ \epsilon} \bullet$ is a cone over $A' \xrightarrow{a} A \xleftarrow{B'} \bullet$, so by pullback $B'[a]$ there is a unique morphism, call it $t[a]$, from family $B[a]$ to $B'[a]$ such that $\epsilon' \circ (t[a]) = t \circ \epsilon$. Thus we have defined $t[a]$ and derived the expected substitution rule:

$$\frac{A \vdash t : B \to B' \qquad \vdash a : A' \to A}{A' \vdash t[a] : B[a] \to B'[a]} \tag{78}$$

In other words, pulling back along a gives rise to a functor $\_[a] : \mathcal{C}/A \to \mathcal{C}/A'$. (This functor is often written in prefix notation as a^* or $a^\sharp$.)

Since substitution is an explicit semantic operation, we must have the semantic equivalent of dummy variables. If A and A' are types, then A' gives rise to a family over A, by means of the following pullback over 1.

$$\begin{tikzcd} \bullet \arrow[r] \arrow[d, "{A'[A]}"'] & A' \arrow[d, "A'"] \\ A \arrow[r, "A"'] & 1 \end{tikzcd} \tag{79}$$

Then weakening is a subcase of substitution, eg:

$$\frac{\vdash a' : A' \qquad \vdash A\ \mathit{Type}}{A \vdash a'[A] : A'[A]} \qquad [\text{informally} : \ \frac{\vdash a' : A' \qquad \vdash A\ \mathit{Type}}{x : A \vdash a' : A'}] \tag{80}$$

1.4 Σ types

If $\mathcal{C}$ has pullbacks, then for $A \in \mathcal{C}$, the left adjoint of $\_[A] : \mathcal{C} \to \mathcal{C}/A$, written $\Sigma(A, \_) : \mathcal{C}/A \to \mathcal{C}$ exists and can be defined as:

$$\vdash \Sigma(A, B) \equiv \mathit{dom}(B)\ \mathit{Type} \tag{81}$$

for $A \vdash B\ \mathit{Type}$. For $a : X \to A$ and $A \vdash B\ \mathit{Type}$ we can now write the pullback square as:

$$\begin{tikzcd} \bullet \arrow[r, "\epsilon"] \arrow[d, "{B[a]}"'] & \Sigma(A,B) \arrow[d, "B"] \\ X \arrow[r, "a"'] & A \end{tikzcd} \tag{82}$$

and if $X \vdash b : B[a]$, we have

$$\begin{tikzcd} X \arrow[r, "b"] \arrow[dr] & \bullet \arrow[r, "\epsilon"] \arrow[d, "{B[a]}"] & \Sigma(A,B) \arrow[d, "B"] \\ & X \arrow[r, "a"'] & A \end{tikzcd} \tag{83}$$

so we can write

$$\vdash \mathit{pair}(a,b) \equiv \epsilon \circ b : X \to \Sigma(A,B) \tag{84}$$

Conversely, if $\vdash p : X \to \Sigma(A,B)$, we write

$$\vdash \pi(p) \equiv B \circ p : X \to A \tag{85}$$

and write $X \vdash \pi'(p) : B[\pi(p)]$ for the unique morphism from the cone $X \xleftarrow{id} X \xrightarrow{p} \Sigma(A,B)$ over the pullback

$$\begin{tikzcd} \bullet \arrow[r] \arrow[d, "{B[\pi(p)]}"'] & \Sigma(A,B) \arrow[d, "B"] \\ X \arrow[r, "\pi(p)"'] & A \end{tikzcd} \tag{86}$$

Given these definitions, the following rules (including surjective pairing) hold.

$$\frac{A \vdash B\ \mathit{Type}}{\vdash \Sigma(A,B)\ \mathit{Type}} \qquad \frac{\vdash a : X \to A \quad X \vdash b : B[a]}{\vdash \mathit{pair}(a,b) : X \to \Sigma(A,B)}$$

$$\frac{\vdash p : X \to \Sigma(A,B)}{\vdash \pi(p) : X \to A, \quad X \vdash \pi'(p) : B[\pi(p)]} \qquad \frac{\vdash a : X \to A \quad X \vdash b : B[a]}{\vdash \pi(\mathit{pair}(a,b)) = a : X \to A}$$

$$\frac{\vdash a : X \to A \quad X \vdash b : B[a]}{X \vdash \pi'(\mathit{pair}(a,b)) = b : B[a]} \qquad \frac{\vdash p : X \to \Sigma(A,B)}{\vdash \mathit{pair}(\pi(p), \pi'(p)) = p : X \to \Sigma(A,B)}$$

Abbreviations. We often abbreviate the first projection as: $\vdash \pi \equiv \pi(id_{\Sigma(A,B)}) : \Sigma(A,B) \to A$. We also abbreviate the functor that substitutes a pair: $\_[\mathit{pair}(a,b)]$ as simply $\_[a,b]$.

1.5 Binary products derived from Σ types

For objects X and B, there is an isomorphism between hom sets $X \to B$ (in $\mathcal{C}$) and $1 \to B[X]$ (in $\mathcal{C}/X$), which we denote:

$$\frac{\vdash b : X \to B}{X \vdash map(b) : B[X]} \quad \text{and} \quad \frac{X \vdash b : B[X]}{\vdash map^{-1}(b) : X \to B} \tag{87}$$

With these we can derive the rules for binary products from the more general rules for Σ types. For $\vdash A, B\ Type$, write:

$$\vdash A \times B \equiv \Sigma(A, B[A])\ Type, \tag{88}$$

and

$$\vdash \pi_1 \equiv \pi : A \times B \to A \quad \text{and} \quad \vdash \pi_2 \equiv map^{-1}(\pi'(id)) : A \times B \to B \tag{89}$$

For $a : X \to A$ and $b : X \to B$, write

$$\vdash \langle a, b \rangle \equiv pair(a, map(b)) : X \to A \times B \tag{90}$$

These definitions use the fact that $B[A][a] = B[A \circ a] = B[X]$. The derived rules for binary products are the following.

$$\frac{\vdash A, B\ Type}{\vdash A \times B\ Type} \qquad \frac{\vdash a : X \to A \quad \vdash b : X \to B}{\vdash \langle a, b \rangle : X \to A \times B}$$

$$\frac{}{\vdash \pi_1 : A \times B \to A, \quad \vdash \pi_2 : A \times B \to B} \qquad \frac{\vdash a : X \to A \quad \vdash b : X \to B}{\vdash \pi_1 \circ \langle a, b \rangle = a : X \to A}$$

$$\frac{\vdash a : X \to A \quad \vdash b : X \to B}{\vdash \pi_2 \circ \langle a, b \rangle = b : X \to B} \qquad \frac{\vdash p : X \to A \times B}{\vdash \langle \pi_1 \circ p, \pi_2 \circ p \rangle = p : X \to A \times B}$$

1.6 Π types and locally cartesian closed categories

To model Π types, for each $A \in \mathcal{C}$ the functor $\_[A] : \mathcal{C} \to \mathcal{C}/A$ must have a right adjoint, which we write as $\Pi(A, \_) : \mathcal{C}/A \to \mathcal{C}$. This is a non-trivial additional requirement, and given that $\mathcal{C}$ has finite limits, it is equivalent to requiring each $\mathcal{C}/A$ be cartesian closed. Thus we arrive at our basic class of models:

Definition 2 *A* locally cartesian closed category *(*Lccc*) $\mathcal{C}$ is a category with finite limits such that each slice category $\mathcal{C}/A$ is cartesian closed.*

If $\mathcal{C}$ is a Lccc, then so is $\mathcal{C}/A$. This gives a natural notion of "left context": rules are given in terms of an arbitrary Lccc, which can be thought of as the given base category or any slice of it.

We can write the rules for the Π type in one of the standard forms for denoting an adjunction: as a co-universal mapping. Here the counit is evaluation and the transpose is currying. Thus we have the following rules:

$$\frac{A \vdash B\ \mathit{Type}}{\vdash \Pi(A,B)\ \mathit{Type}} \quad \frac{A \vdash B\ \mathit{Type}}{A \vdash \mathit{eval} : \Pi(A,B)[A] \to B} \quad \frac{A \vdash f : X[A] \to B}{\vdash \lambda(f) : X \to \Pi(A,B)}$$

$$\frac{A \vdash f : X[A] \to B}{A \vdash \mathit{eval} \circ \lambda(f)[A] = f : X[A] \to B} \quad \frac{\vdash g : X \to \Pi(A,B)}{\vdash \lambda(\mathit{eval} \circ g[A]) = g : X \to \Pi(A,B)}$$

(Note: the last rule is "η".) Letting X be 1, then $1[A] = 1$, and currying plus substitution derives the β rule:

$$\frac{A \vdash f : B \qquad \vdash a : A}{\vdash \mathit{eval}[a] \circ \lambda(f) = f[a] : B[a]} \tag{91}$$

The exponent type $A \Rightarrow B$ can be defined in terms of Π, in the same way binary product was defined in terms of Σ.

1.7 Identity types

Identity types can be modeled in any Lccc. For $a, a' : X \to A$, define $X \vdash I(A, a, a')\ \mathit{Type}$ as the equalizer of a and a'. Given $X \vdash B\ \mathit{Type}$, if $\vdash a \circ \pi = a' \circ \pi : \Sigma(X,B) \to A$, then there is a unique morphism, call it *true*, in the hom set $B \to I(A, a, a')$. (And as $I(A, a, a')$ is a monic morphism, any hom set $B \to I(A, a, a')$ consists of at most one morphism.) Conversely, if there is a morphism in this hom set, then by arrow chasing, $\vdash a \circ \pi = a' \circ \pi : \Sigma(X,B) \to A$. Thus we have the rules:

$$\frac{\vdash a, a' : X \to A}{X \vdash I(A, a, a')\ \mathit{Type}} \qquad \frac{X \vdash B\ \mathit{Type} \qquad \vdash a \circ \pi = a' \circ \pi : \Sigma(X,B) \to A}{X \vdash \mathit{true} : B \to I(A, a, a')}$$

$$\frac{X \vdash t : B \to I(A, a, a')}{\vdash a \circ \pi = a' \circ \pi : \Sigma(X,B) \to A} \quad \frac{X \vdash t : B \to I(A, a, a')}{X \vdash \mathit{true} = t : B \to I(A, a, a')}$$

In the special case $B = 1$, using the fact that $\pi : \Sigma(X, 1) \to X$ is an isomorphism, we can derive the rules:

$$\frac{\vdash a = a' : X \to A}{X \vdash \mathit{true} : I(A, a, a')} \quad \frac{X \vdash t : I(A, a, a')}{\vdash a = a' : X \to A} \quad \frac{X \vdash t : I(A, a, a')}{X \vdash \mathit{true} = t : I(A, a, a')}$$

1.8 The empty type and $A + B$

If $\mathcal{C}$ has finite coproducts, then we can model the empty type as the initial object and $A + B$ as the coproduct of A and B. Note: if Lccc $\mathcal{C}$ has finite coproducts so does each $\mathcal{C}/A$. Thus we have the expected rules:

$$\frac{}{\vdash 0\ \mathit{Type}} \qquad \frac{\vdash X\ \mathit{Type}}{\vdash \mathit{init} : 0 \to X} \qquad \frac{\vdash t : 0 \to X}{\vdash t = \mathit{init} : 0 \to X}$$

$$\frac{\vdash A, B\ \mathit{Type}}{\vdash A + B\ \mathit{Type}} \qquad \frac{\vdash a : A \to X \quad \vdash b : A \to X}{\vdash [a, b] : A + B \to X}$$

$$\frac{}{\vdash \mathit{inl} : A \to A + B} \qquad \frac{}{\vdash \mathit{inr} : B \to A + B}$$

$$\frac{\vdash a : A \to X \quad \vdash b : A \to X}{\vdash [a, b] \circ \mathit{inl} = a : A \to X} \qquad \frac{\vdash a : A \to X \quad \vdash b : A \to X}{\vdash [a, b] \circ \mathit{inr} = b : B \to X}$$

$$\frac{\vdash t : A + B \to X}{\vdash [t \circ \mathit{inl}, t \circ \mathit{inr}] = t : A + B \to X}$$

One can use $[\_, \_]$ to define the seemingly more elaborate type theoretic elimination form. We will demonstrate the general technique for doing this next, with the natural numbers.

1.9 The natural numbers, via initial algebras

Recall the definition of a natural numbers object: a triple $(N, 0, S)$ such that N is an object, $0 : 1 \to N$ and $S : N \to N$, and for any other such triple (A, a, b), there is a unique morphism $R : N \to A$ making the following diagram commute. (Don't confuse the element 0 with the initial object.)

$$\begin{array}{ccccc} & 0 & & S & \\ 1 & \longrightarrow & N & \longrightarrow & N \\ \downarrow & & \downarrow R & & \downarrow R \\ 1 & \longrightarrow & A & \longrightarrow & A \\ & a & & b & \end{array} \tag{92}$$

If a category has binary coproducts, this is equivalent to $[0, S] : 1 + N \to N$ being the initial Φ-algebra for the functor $\Phi(X) = 1 + X$.

All well and good, that, but what if we want to define inductively an element R' of the type family $A \in \mathcal{C}/N$? This seemingly more elaborate morphism can be defined using only the properties of the natural numbers object, as we now demonstrate.

Suppose we are given a Φ-algebra $[a, b] : 1 + \Sigma(N, A) \to \Sigma(N, A)$ such that the morphism $\pi : \Sigma(N, A) \to N$ is a homomorphism from $[a, b]$ to $[0, S]$, ie, the following diagram commutes.

$$\begin{array}{ccc} & [a, b] & \\ 1 + \Sigma(N, A) & \longrightarrow & \Sigma(N, A) \\ 1 + \pi \quad \downarrow & & \downarrow \quad \pi \\ 1 + N & \longrightarrow & N \\ & [0, S] & \end{array} \tag{93}$$

By surjective pairing, let $a = pair(a_1, a_2)$ and $b = pair(b_1, b_2)$. That (93) commutes is equivalent to the following four conditions.

$$\begin{array}{ll} \vdash a_1 = 0 : 1 \to N & \vdash b_1 = S \circ \pi : \Sigma(N, A) \to N \\ \vdash a_2 : A[0] & X \vdash b_2[n, a'] : A[Sn] \end{array} \tag{94}$$

for an arbitrary $pair(n, a') : X \to \Sigma(N, A)$. Now $[0, S]$ is still an initial Φ-algebra, so there is a unique morphism $R : N \to \Sigma(N, A)$ such that $R \circ [0, S] = [a, b] \circ (1 + R)$. Furthermore, by uniqueness of the homomorphism, $\pi \circ R = id : N \to N$, ie, R is of the form $pair(id, R')$, for some $N \vdash R' : A$. R being a homomorphism is equivalent to requiring of R' that:

$$\begin{array}{l} \vdash R'[0] = a_2 : A[0] \\ X \vdash R'[Sn] = b_2[n, R'[n]] : A[Sn] \end{array} \tag{95}$$

for an arbitrary $n : X \to N$. Thus we have derived the usual "dependent type" form of the induction rule for the natural numbers, and have shown R' is the unique morphism satisfying (95). Note that our approach here is completely general, and can be applied uniformly to other inductive types: W types [6] and "tree types" [9]. Dualizing to final coalgebras yields coinductive, or non-well-founded types.

2 Coequalizers and quotient types

Recall that a *coequalizer* of a parallel pair of morphisms $f, f' : B \to A$ is a morphism $c : A \to C$ such that $c \circ f = c \circ f'$, and for any other morphism $c' : A \to C'$ such that $c' \circ f = c' \circ f'$ there is a unique $u : C \to C'$ making $u \circ c = c'$. Such a c is necessarily epic. If $\mathcal{C}$ has coequalizers, so does $\mathcal{C}/A$, as they are created by the forgetful functor.

We now turn our attention to coequalizers, so assume for the rest of this note, that $\mathcal{C}$ is a locally cartesian closed category with coequalizers.

Motivation. In category Set, a pair of functions $f, f' : B \to A$ defines a predicate $R \subseteq A \times A$:

$$R(x, y) \iff \exists b \in B.\ f(b) = x \wedge f'(b) = y \tag{96}$$

A coequalizer of f and f' (notation: $q : A \to A/(f, f')$) quotients A by the least equivalence relation containing R. Carrying out this in an arbitrary Lccc with coequalizers yields the following rules, which we follow with their explanation.

$$\frac{\vdash f, f' : B \to A}{\vdash A/(f, f')\ Type} \quad \frac{\vdash f, f' : B \to A}{\vdash q : A \to A/(f, f')} \quad \frac{\vdash f, f' : B \to A}{\vdash q \circ f = q \circ f' : B \to A/(f, f')}$$

$$\frac{A/(f, f') \vdash C\ Type \quad A \vdash c : C[q] \quad B \vdash c[f] = c[f'] : C[q \circ f]}{A/(f, f') \vdash quo(c) : C, \quad A \vdash quo(c)[q] = c : C[q]}$$

$$\frac{A(f/f') \vdash d : C}{A/(f, f') \vdash quo(d[q]) = d : C}$$

Formation, equality. Given $f, f : B \to A$, define $q : A \to A/(f, f')$ as the coequalizer of f and f'.

Elimination. We have the pullback square

$$\begin{tikzcd} \bullet \arrow[r, "\epsilon"] \arrow[d, "{C[q]}"'] & \bullet \arrow[d, "C"] \\ A \arrow[r, "q"'] & A/(f, f') \end{tikzcd} \tag{97}$$

and $B \vdash c[f] = c[f'] : C[q \circ f]$ is equivalent to $\epsilon \circ c \circ f = \epsilon \circ c \circ f'$. Thus there is a unique $quo(c)$ such that

$$\begin{tikzcd} A \arrow[r, "q"] \arrow[d, "c"'] & A/(f, f') \arrow[d, "quo(c)"] \\ \bullet \arrow[r, "\epsilon"'] & \bullet \end{tikzcd} \tag{98}$$

commutes. q being epic implies $C \circ quo(c) = id$, so that $A/(f, f') \vdash quo(c) : C$. Thus $quo(c)$ is the unique element in $1 \to C$ such that $quo(c)[q] = c$.

2.1 Example: internalizing epimorphisms

It's easy to show that a morphism $f : X \to Y$ in $\mathcal{C}$ is monic iff the type corresponding to the statement of injectivity:

$$(\forall x, x' \in X) f(x) = f(x') \Rightarrow x = x' \tag{99}$$

is inhabited (ie, has a global element). Equivalently, one can show there is a morphism in the hom set $m \to \Delta$ of $\mathcal{C}/(X \times X)$, where $m : M \to X \times X$ is the equalizer of $f \circ \pi_1, f \circ \pi_2 : X \times X \to Y$, or that the type corresponding to the statement

$$(\exists h \in M \to X) m = \Delta \circ h \tag{100}$$

is inhabited.

However, in a locally cartesian closed category, the statement of surjectivity:

$$(\forall y \in Y)(\exists x \in X) f(x) = y \tag{101}$$

is equivalent to f being a *split* epi — contrast that with the situation in a topos [4]. Of course, using coequalizers, we can simply dualize the earlier argument — specifically, we dualize (100). (We can't dualize (99) directly in the type theory, because of the difficulty in dualizing the notion of "family of types.") So $f : X \to Y$ is epic iff the type corresponding to the statement

$$(\exists h \in Y \to E) q = h \circ \nabla \tag{102}$$

is inhabited, where $E \equiv (Y + Y)/(inl \circ f, inr \circ f)$, and $q : Y + Y \to E$ is the coequalizing morphism, and ∇ is $[id, id] : Y + Y \to Y$.

2.2 Example: the "squash" type

Call an object B *partially final* (better name please!) if for any C the hom set $C \to B$ contains *at most* one morphism.

The "squash" of a type A, written $\Diamond A$, is supposed to be A "without the computational content." In the term model definition we say $\Diamond A$ is inhabited by some unique closed term iff A is inhabited by any closed term, but this doesn't generalize to a categorical description. However, an elimination rule provides the insight: there is a map $nec : A \to \Diamond A$ and for every "constant" morphism $c : A \to C$ (ie, $c(x) = c(y)$ for $x, y \in A$) there is a unique $\flat(c) : \Diamond A \to C$ (geddit?) making $c = \flat(c) \circ nec$. Of course, this is the definition of $nec : A \to \Diamond A$ as the coequalizer of $\pi_1, \pi_2 : A \times A \to A$. We shall see this yields a partially final $\Diamond A$. So take the quotient rules and write $\Diamond A$ for $A/(\pi_1, \pi_2)$,

$nec : A \to \Diamond A$ for q and $\flat(c)$ for $quo(c)$. They now read:

$$\frac{\vdash A\ Type}{\vdash \Diamond A\ Type} \quad \frac{\vdash A\ Type}{\vdash nec : A \to \Diamond A} \quad \frac{\vdash A\ Type}{\vdash nec \circ \pi_1 = nec \circ \pi_2 : A \times A \to \Diamond A}$$

$$\frac{\Diamond A \vdash C\ Type \quad A \vdash c : C[nec] \quad A \times A \vdash c[\pi_1] = c[\pi_2] : C[nec \circ \pi_1]}{\Diamond A \vdash \flat(c) : C, \quad A \vdash \flat(c)[nec] = c : C[nec]} \quad \frac{\Diamond A \vdash d : C}{\Diamond A \vdash \flat(d[nec]) = d : C}$$

Sequent $A \times A \vdash c[\pi_1] = c[\pi_2] : C[nec \circ \pi_1]$ is informally stating "$x : A, y : A \vdash c(x) = c(y) : C(nec(x))$" and the NuPRL approach of requiring $x : A$ not to occur in c guarantees this. Lemma 1 shows $\Diamond A$ is partially final, ie:

$$\frac{\vdash a, a' : C \to \Diamond A}{\vdash a = a' : C \to \Diamond A} \tag{103}$$

Lemma 1 *1. $B \in \mathcal{C}$ is partially final iff $\pi_1 = \pi_2 : B \times B \to B$ iff $nec : B \to \Diamond B$ is an isomorphism.*

2. For any A in LCCC *$\mathcal{C}$, $\Diamond A$ is partially final.*

Proof. Assertion 1 is easy. To show $\Diamond A$ is partially final, first note that the functors $A \times \_$ and $\_ \times \Diamond A$, being left adjoints, preserve all colimits, in particular coequalizers. Thus

$$A \times nec : A \times A \to A \times \Diamond A \quad \text{and} \quad nec \times \Diamond A : A \times \Diamond A \to \Diamond A \times \Diamond A$$

are coequalizing morphisms, and so are epis, and thus their composition, $nec \times nec$, is an epi. Let π_1 and π_2 be the projections from $A \times A$ and p_1 and p_2 be the projections from $\Diamond A \times \Diamond A$. It suffices to show $p_1 = p_2$, but that follows from: $p_1 \circ (nec \times nec) = nec \circ \pi_1 = nec \circ \pi_2 = p_2 \circ (nec \times nec)$, and $nec \times nec$ being epic. $\Box$

Squashing the identity, negation and squash types. We think of $I(A, a, a')$, $\neg A$ and $\Diamond A$ as having no "computational content," so squashing them ought to be a redundant operation. This is because they are already partially final. We have already noted the partial finality of identity types and the squash type itself, and for negation, recall that 0 is partially final in any bicartesian closed category. Since $C \to (\neg A) \cong (C \times A) \to 0$, $\neg A$ is partially final. Now apply assertion 1 of lemma 1 to find $\Diamond\neg A \cong \neg A$.

2.3 Example: a weak subset type

For $A \vdash B$ *Type*, we can define the notation

$$\vdash \{A|B\} \equiv \Sigma(A, \Diamond B) \; Type \tag{104}$$

Given $\vdash a : X \to A$, then $X \vdash nec[a] : B[a] \to (\Diamond B)[a]$ and thus if $X \vdash b : B[a]$, then $X \vdash nec[a] \circ b : (\Diamond B)[a]$. More generally, for $X \vdash t : (\Diamond B)[a]$, write

$$\vdash in(a) \equiv \langle a, t \rangle : X \to \{A|B\} \tag{105}$$

This is not ambiguous with respect to t because $(\Diamond B)[a]$ is partially final (right adjoints must preserve partially final objects). For $\vdash p : X \to \{A|B\}$, define the notation

$$\vdash out(p) \equiv \pi(p) : X \to A \tag{106}$$

So we have the rules:

$$\frac{A \vdash B \; Type}{\vdash \{A|B\} \; Type} \qquad \frac{\vdash a : X \to A \quad X \vdash b : B[a]}{\vdash in(a) : X \to \{A|B\}} \qquad \frac{\vdash p : X \to \{A|B\}}{\vdash out(p) : X \to A}$$

$$\frac{\vdash a : X \to A \quad X \vdash b : B[a]}{\vdash out(in(a)) = a : X \to A} \quad \frac{\vdash p : X \to \{A|B\}}{\vdash in(out(p)) = p : X \to \{A|B\}}$$

There is also the elimination rule arising from $\Diamond B$ (recall $\{A|B\} \vdash \ldots$ is equivalent to $\Diamond B \vdash \ldots$):

$$\frac{\{A|B\} \vdash C \; Type \quad B \vdash c : C[nec] \quad B \times B \vdash c[\pi_1] = c[\pi_2] : C[nec \circ \pi_1]}{\{A|B\} \vdash \flat(c) : C, \quad B \vdash \flat(c)[nec] = c : C[nec]}$$

$$\frac{\{A|B\} \vdash d : C}{\{A|B\} \vdash \flat(d[nec]) = d : C}$$

The condition $B \times B \vdash c[\pi_1] = c[\pi_2] : C[nec \circ \pi_1]$ is achieved in the NuPrl elimination rule by restricting the variable $y : B$ not to occur in c. But this actually results in a stronger rule than can justified with our approach, because in our semantics we can't express the idea that a value is not used "computationally." The best we can do is to trivialize the value by squashing. This difference in meanings is demonstrated in the PER term model [5, 2], where

$\Sigma(A, \Diamond B)$ corresponds to the initial epi/mono factorization of $\pi : \Sigma(A, B) \to A$, while the NuPrl $\{A|B\}$ type is modeled by what turns out to be the final factorization.

3 Conclusion

It is often advantageous to think of a Martin-Löf type theory as the internal language of a locally cartesian closed category, because then semantics can inspire our proof rules. This note was meant to be an exercise in working from a categorical property (the existence of coequalizers) to arrive at the type theory rules for quotient types. We used the quotient types to define a "squash" type and then a weak subset type. The semantics of the "true" subset type probably requires separating the notions of type and proposition, as is done in [8].

References

[1] R.L. Constable et al. *Implementing Mathematics with the* NUPRL *Proof Development System.* Prentice-Hall, 1986.

[2] Robert Harper. Constructing type systems over an operational semantics. Technical Report ECS-LFCS-88-59, University of Edinburgh, 1988.

[3] J. Martin E. Hyland and Andrew M. Pitts. The theory of constructions: categorical semantics and topos-theoretic models. *Contemporary Mathematics*, 92:137–199, 1989.

[4] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic.* Cambridge studies in advanced mathematics 7. Cambridge University Press, 1986.

[5] G. Longo and E. Moggi. Constructive natural deduction and its "modest" interpretation. In M. Gawron, D. Israel, J. Meseguer, and S. Peters, editors, *Semantics of natural and computer languages.* MIT Press, 1988.

[6] P. Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science.* North-Holland, 1982.

[7] P. Martin-Löf. Intuitionistic type theory. Notes by G. Sambin of a series of lectures given in Padua, June 1980, Bibliopolis, Napoli, 1984.

[8] B. Nordstrom, K. Peterson, and J. Smith. *An Introduction to Martin-Löf's Theory of Types.* Oxford University Press, 1990.

[9] K. Petersson and D. Synek. A set constructor for inductive sets in Martin-Löf's type theory. In *Category Theory and Computer Science*, pages 128–140. Springer, LNCS 389, 1989.

[10] R.A.G. Seely. Local cartesian closed categories and type theory. *Math. Proc. Camb. Phil. Soc.*, 95:33–48, 1984.

A Critical Pair Lemma for Higher-Order Rewrite Systems and its Application to λ^*

(Draft Version)

Tobias Nipkow

University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
England
`tnn@cl.cam.ac.uk`

Abstract

We consider rewrite systems over simply typed λ-terms where the left-hand sides obey the following restriction: in their β-normal form, any occurrence of a free variable x must be of the form $x(y_1, \ldots, y_n)$, where the y_i are distinct bound variables. Calling such terms "patterns" we obtain two important results:

- Unification of two patterns is decidable and single most general unifiers exist. This result is due to Dale Miller.
- The critical pair lemma can be extended from first-order to such restricted higher-order rewrite systems.

This framework can be used to formalize the reduction relation in various typed λ-calculi and obtain confluence results by simple critical pair considerations. This is demonstrated for a particular system, λ^*, which combines product and function types by imposing certain equalities between them, namely all the isomorphisms in a CCC, minus commutativity of products. A potential application of this system to the unification problem for the typed λ-calculus with ML-style polymorphism is sketched.

1 Introduction

In 1972, Knuth and Bendix published their seminal paper [9] which shows that confluence of terminating algebraic rewrite systems is decidable: a simple test of confluence for the finite set of so called critical pairs suffices. The objective of this paper is to generalize this construction from first-order rewrite systems (all functions are first-order) to rewrite systems over simply typed λ-terms. The aim of this generalization is to lift the rich theory developed around first-order rewrite

[1]Research supported by ESPRIT Basic Research Action "Logical Frameworks".

systems and apply it to systems manipulating higher-order terms such as program transformers, theorem provers and the like.

Section 2 reviews the basic notation for typed λ-terms and substitutions. Section 3 introduces a subclass of λ-terms called *patterns* which have a simple unification problem. *Higher-order rewrite systems* are defined to be rewrite systems over typed λ-terms whose left-hand sides are patterns: this guarantees that the rewrite relation is easily computable. In Section 4 the notion of *critical pairs* is generalized to higher-order rewrite systems and the analogue of the critical pair lemma is proved. The restricted nature of patterns is instrumental in obtaining these results. Finally, Section 5 applies the critical pair lemma to a number of λ-calculi formalized as higher-order rewrite systems. Of particular interest is λ^*, a typed system with products which are related to functions by implicit currying and un-currying.

This paper is a draft version of a full paper that is in preparation. Only important definitions and theorems are included and no proofs are provided.

2 Preliminaries

2.1 Terms

What follows is a description of the meta-language of simply typed λ-calculus which will be used to define several object λ-calculi. The notation is roughly consistent with the standard literature [5, 3, 7].

Starting with some fixed set of *base types* $\mathcal{B}$, the set $\mathcal{T}$ of all *types* is the closure of $\mathcal{B}$ under the function space constructor "o". The letter τ represents types. Function types associate to the right: $\tau_1 o \tau_1 o \tau_3$ means $\tau_1 o (\tau_2 o \tau_3)$. Instead of $\tau_1 o \cdots o \tau_n o \tau$ we write $\overline{\tau_m} o \tau$. The latter form is used only if τ is a base type.

Terms are generated from a set of typed *variables* $V = \bigcup_{\tau \in \mathcal{T}} V_\tau$ and a set of typed *constants* $C = \bigcup_{\tau \in \mathcal{T}} C_\tau$, where $V_\tau \cap V_{\tau'} = C_\tau \cap C_{\tau'} = \{\}$ if $\tau \neq \tau'$, by λ-abstraction and application. Variables are denoted by x, y and z, and constants by a, b, and c. Terms are denoted by l, r, s, t, and u. The inductive definition of *typed terms* is as follows:

$$\frac{x \in V_\tau}{x : \tau} \qquad\qquad \frac{c \in C_\tau}{c : \tau}$$

$$\frac{s : \tau o \tau' \quad t : \tau}{(s\ t) : \tau'} \qquad\qquad \frac{x : \tau \quad s : \tau'}{([x]s) : \tau o \tau'}$$

We write $[x]s$ for the meta-level abstraction rather than the more conventional $\lambda x.s$ which will be used for object level abstraction. Instead of $[x_1]\ldots[x_n]s$ we write $[x_1, \ldots, x_n]s$ or just $[\overline{x_n}]s$. Similarly we write $t(u_1, \ldots, u_n)$ or just $t(\overline{u_n})$ instead of $(\ldots(t\ u_1)\ldots)u_n$. The *free* and *bound* variables occurring in a term are denoted by $\mathcal{FV}(s)$ and $\mathcal{BV}(s)$, respectively.

We assume the usual definition of α, β and η conversion [5] between λ-terms. We write $s =_\gamma t$, where $\gamma \in \{\alpha, \beta, \eta\}$ if s and t are equivalent modulo γ-conversion, and $s \equiv t$ iff s and t are equivalent modulo α, β and η conversion. We write $s \longrightarrow_\beta t$ if t is the result of a single β-reduction is s.

The simply typed λ-calculus is confluent and terminating w.r.t. β-reduction and the *β-normal form* of a term t is denoted by $t{\downarrow_\beta}$. The *η-expanded form* of a term $t = [\overline{x_n}]c(\overline{u_m})$ in β-normal form is defined by

$$t{\downarrow_\eta} = [\overline{x_{n+k}}]c(\overline{u_m{\downarrow_\eta}}, x_{n+1}{\downarrow_\eta}, \ldots, x_{n+k}{\downarrow_\eta})$$

where $t : \overline{\tau_{n+k}} o \tau$ and $x_{n+1}, \ldots, x_{n+k} \notin \mathcal{FV}(\overline{u_m})$. Instead of $t{\downarrow_\beta}{\downarrow_\eta}$ we write $t{\downarrow}$ or $\hat{t}$. A term t is in *normal form* if $t = \hat{t}$. It is well known [17] that $s \equiv t$ iff $\hat{s} =_\alpha \hat{t}$. A term in normal form must be of the form $[\overline{x_k}]h(\overline{u_n})$ such that h is a constant or variable, called the *head* of the term.

Substitutions are finite mappings from variables to terms of the same type. Substitutions are denoted by θ. If $\theta = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define $\mathcal{D}om(\theta) = \{x_1, \ldots, x_n\}$, $\mathcal{R}an(\theta) = \{t_1, \ldots, t_n\}$. We distinguish two forms of applications of substitutions to terms

$$\theta[t]{=}[t_1/x_1, \ldots, t_n/x_n]t$$
$$\theta(t){=}([\overline{x_n}]t)(\overline{t_n}){\downarrow} = \theta[t]{\downarrow}$$

where $[t_1/x_1, \ldots, t_n/x_n]t$ is the *simultaneous* substitution of the t_i for the x_i (including any renaming of bound variables). These definitions are independent of the order of the $x_i \mapsto t_i$ pairs (see Huet [6]). The composition of two substitutions θ and θ' is defined as $(\theta' \circ \theta)(t) = \theta'(\theta(t))$ and $(\theta' \circ \theta)[t] = \theta'[\theta[t]]$. The restriction of θ to some subset W of its domain is denoted by $\theta|_W$. Overwriting is defined as $\theta + \theta' = \theta|_{\mathcal{D}om(\theta) - \mathcal{D}om(\theta')} \cup \theta'$. Given a binary relation $\cong$ on terms, $\theta \cong \theta'$ means $\mathcal{D}om(\theta') = \mathcal{D}om(\theta)$ and $\theta' x \cong \theta x$ for all x. Two substitutions θ and θ' are *equal over W*, written $\theta \equiv \theta'\ [W]$, iff $\theta x \equiv \theta' x$ holds for all $x \in W$.

A *renaming* ρ is an injective substitution with $\mathcal{R}an(\rho) \subset V$. Renamings are always denoted by ρ.

Terms can also be viewed as trees. Subterms can be numbered by so-called *positions* which are the paths from the root to the subterm in Dewey decimal notation. Details can be found in [7, 3]. We just briefly review the notation. The *positions* in a term t are denoted by $\mathcal{P}os(t) \subseteq \mathbb{N}^*$. The letters p and q stand for positions. The root position is ε, the empty sequence. Two positions p and q are appended by juxtaposing them: pq. Note that natural numbers are valid positions. We write $p \leq q$ if p is a prefix of q. In that case there is a p' such that $pp' = q$ and q/p is defined as the suffix p'. If neither $p \leq q$ nor $p \geq q$, we write $p \parallel q$, indicating that p and q are in different subtrees. Given $p \in \mathcal{P}os(t)$, t/p is the subterm of t at position p; $t[u]_p$ is t with t/p replaced by u.

Abstractions and applications yield the following trees:

$$\begin{array}{c} [x] \\ | \\ s \end{array} \qquad\qquad \begin{array}{c} \cdot \\ /\ \backslash \\ s \quad t \end{array}$$

Hence positions in λ terms are sequences over $\{1, 2\}$. Note that the bound variable in an abstraction is not a separate subterm and can therefore not be accessed by the s/p notation.

It has to be stressed that we do *not* work with α-equivalence classes of terms. Otherwise the notation s/p would not make sense because $=_\alpha$ is not a congruence w.r.t. $/$: $s_1 =_\alpha s_2$ does not imply $s_1/p =_\alpha s_2/p$ because s_i/p may contain free variables which are bound in s_i.

Given a relation $\longrightarrow$, $\longrightarrow^*$ denotes the transitive and reflexive closure of $\longrightarrow$. We write $s \downarrow t$ iff there is a u such that $s \longrightarrow^* u$ and $t \longrightarrow^* u$. The relation $\longrightarrow$ is *(locally) confluent* if $r \longrightarrow^* s$ ($r \longrightarrow s$) and $r \longrightarrow^* t$ ($r \longrightarrow t$) imply $s \downarrow t$. The relation $\longrightarrow$ is *terminating* if there is no infinite sequence $s_i \longrightarrow s_{i+1}$ for all $i \in \mathbb{N}$.

3 Higher-Order Rewrite Systems

Higher-Order Rewrite Systems (HRS) are similar to Klop's *Combinatory Reduction Systems (CRS)* [8]. Both are generalizations of first-order rewrite systems [3] to terms with higher-order functions and bound variables. The main difference is that HRS use the typed and CRS the untyped λ-calculus as a meta-language.

Definition 3.1 A term t in normal form is called a *pattern for* x, $x : \overline{\tau_n} o \tau$, if every free occurrence of x in t is in a subterm $x(\overline{u_n})$ such that $\overline{u_n}$ is η-equivalent to a list of distinct bound variables; t is called a *pattern* if it is a pattern for all its free variables.

Examples of patterns are $[x]c(x)$, y, $[x]y([z]x(z))$, and $[x, y]z(y, x)$. Examples of non-patterns are $x(c)$, $[x]y(x, x)$ and $[x]z(z(x))$.

The following crucial result about unification of patterns is due to Dale Miller [10]:

Theorem 3.1 *It is decidable whether two patterns are unifiable; if they are unifiable, a most general unifier can be computed.*

For the algorithm itself the reader is referred to [10].

This result will ensure both the computability of the rewrite relation defined by a HRS and of critical pairs.

In the sequel we have to deal with conditional rewrite systems. The idea to encode the syntax of logical systems in the typed λ-calculus goes back to Church. More recently it has been taken up again by Martin-Löf [12]. Hence we simply assume conditions are λ-terms of some distinguished type o of *propositions*. Propositions are denoted by P and Q and come with a proof system $\vdash$. We assume the existence of the basic logical constants $\Longrightarrow$, $\wedge$, and $\bigwedge$ (universal quantification) on propositions and that provability is

- invariant under α, β and η equality and under renamings,
- preserved under substitutions (free variables are universally quantified).

An example of such a system is Church's Higher-Order Logic as found for example in HOL and Isabelle [4, 14].

Definition 3.2 A *rewrite rule* is a conditional equality $P \Longrightarrow l \longrightarrow r$ such that l is a pattern but not η-equivalent to a free variable, l and r are of the same type, and $\mathcal{FV}(l) \supseteq \mathcal{FV}(r)$. A

Higher-Order Rewrite System (for short: *HRS*) is a finite set of rewrite rules. The letter R always denotes a HRS. A HRS R induces a relation $\longrightarrow_R$ on terms:

$$s \longrightarrow_R t \quad \Leftrightarrow \quad \exists (P \Longrightarrow l \longrightarrow r) \in R, p \in \mathcal{P}os(\hat{s}), \theta.\ (\vdash \theta P) \ \wedge \ \hat{s}/p \equiv \theta l \ \wedge \ t \equiv \hat{s}[\theta r]_p.$$

Remarks:

- $\longrightarrow_R$ is invariant under $\equiv$: $s' \equiv s \longrightarrow_R t \equiv t'$ implies $s' \longrightarrow_R t'$.
- Left-hand sides of rules are by definition in normal form.
- $\hat{s}/p \equiv \theta l$ is equivalent to $\hat{s}/p =_\alpha \theta l$.
- $\longrightarrow_R$ enjoys a weak from of *compatibility*: $t \longrightarrow_R u$ implies $\hat{s}[t]_p \longrightarrow_R \hat{s}[u]_p$.

Although this definition of rewriting is very restrictive (only subterms of terms in normal form may be rewritten) it is not even obviously decidable[1] since it requires higher-order matching of subterms with right-hand sides of rules. Fortunately, Theorem 3.1 tells us that this is decidable for patterns.

In addition to the rather operational $\longrightarrow_R$ we have the logical notion of equality modulo R. The latter is formalized by taking α, β, and η-conversion together with all instances of R

$$\{\theta r = \theta s \mid \exists (P \Longrightarrow r \longrightarrow s) \in R, \theta.\ \vdash \theta P\}$$

as axioms and closing under reflexivity, symmetry, transitivity and the two congruence laws

$$\frac{s = t}{[x]s = [x]t} \qquad \frac{s = s' \quad t = t'}{(s\ t) = (s'\ t')}.$$

The resulting relation is called $=_R$. The following theorem shows that, modulo $\equiv$, $\longrightarrow_R^*$ coincides with $=_R$:

Theorem 3.2 *If all rules in R are of ground type, then*

$$s =_R t \quad \Leftrightarrow \quad (s \longleftrightarrow_R^* t \ \vee \ s \equiv t)$$

Notice that this equivalence between $=_R$ and $\longleftrightarrow_R^*$ does only work for rules of base type. Given the rule

$$[x]c(x, y(x)) \ \longrightarrow \ y$$

the relation $\longleftrightarrow_R^*$ is strictly weaker than $=_R$: although we have $c(a, y(a)) =_R y$, $c(a, y(a)) \longrightarrow_R y$ does not hold. The reason is the restrictive definition of $\longrightarrow_R$ which insists on rewriting only terms in β-normal form. Otherwise it would be easy to rewrite $c(a, y(a)) \equiv ([x]c(x, y(x)))(a)$. The restriction to the simpler form of rewriting is due to technical problems inherent in the more general notion: it is less obvious how to decide whether a rewrite rule is applicable, and the proof of the critical pair lemma would be complicated further.

[1] Even if $\vdash$ is decidable.

4 The Critical Pair Lemma

For first-order rewrite systems, critical pairs arise from unifying the left-hand side of one rule with the subterm of the left-hand side of another rule:

$$(x \times y) \times z \longrightarrow x \times (y \times z)$$
$$i(x \times y) \longrightarrow i(y) \times i(x)$$

gives rise to two critical pairs, one of which is the result of reducing $i((x \times y) \times z)$ in two different ways:

$$\begin{tikzcd} i((x \times y) \times z) \arrow[r] \arrow[d] & i(z) \times i(x \times y) \arrow[d, dashed, "*"] \\ i(x \times (y \times z)) \arrow[r, dashed, "*"] & i(z) \times (i(y) \times i(x)) \end{tikzcd}$$

The dashed arrows indicate the common reduct of the critical pair.

Due to the presence of bound variables, critical pairs for HRS are more complex:

Definition 4.1 Let $P_i \Longrightarrow l_i \longrightarrow r_i$, $i = 1, 2$, be two rules such that $\mathcal{FV}(l_1) \cap \mathcal{BV}(l_1) = \{\}$, let $p \in \mathcal{P}os(l_1)$ such that the head of l_1/p is not a free variable, let $\overline{x_k} = \mathcal{FV}(l_1/p) \cap \mathcal{BV}(l_1)$ where $x_i : \tau_i$, let ρ be a renaming such that $\mathcal{D}om(\rho) = \mathcal{FV}(l_2)$, $\mathcal{R}an(\rho) \cap \mathcal{FV}(l_1) = \{\}$, and $\rho(x) : \tau_1 o \cdots o \tau_k o \tau$ if $x : \tau$, let $\sigma = \{x \mapsto \rho(x)(\overline{x_k}) \mid x \in \mathcal{FV}(l_2)\}$, and let θ be a most general unifier of $[\overline{x_k}]l_1/p$ and $[\overline{x_k}]\sigma l_2$. Then

$$\theta(P_1 \wedge (\bigwedge [\overline{x_k}]\sigma P_2) \implies r_1 = l_1[\sigma r_2]_p)$$

is called a *critical pair*.

A proposition P is called *R-closed* if $\vdash \theta P$ and $\theta \longrightarrow_R^* \theta'$ imply $\vdash \theta' P$.

Lemma 4.1 (Critical Pair Lemma) *Let R be a HRS where all rules are of base type and all preconditions are R-closed. If $s \longrightarrow_R t_1$ and $s \longrightarrow_R t_2$, then either $t_1 \downarrow_R t_2$, or there are a critical pair $P \Longrightarrow u_1 = u_2$, a substitution δ and a position $p_1 \in \mathcal{P}os(\hat{s})$ such that $\vdash \delta P$ and $t_i \equiv \hat{s}[\delta u_i]_{p_1}$ for $i = 1, 2$.*

This leads to the following very general corollary:

Corollary 4.1 *Let R be a HRS where all rules are of base type and all preconditions are R-closed, and let S be some subset of all terms which is closed under $\longrightarrow_R$, $\equiv$, and under taking subterms of base type. If $\theta u_1 \downarrow_R \theta u_2$ for all critical pairs $P \Longrightarrow u_1 = u_2$ and all θ such that $\theta u_1, \theta u_2 \in S$ and $\vdash \theta P$, then $\longrightarrow_R$ is locally confluent on S.*

For unconditional HRS there is a further corollary which resembles the corresponding theorem for first order term-rewriting systems.

Corollary 4.2 *Let R be a HRS where all rules are of base type. If $u_1 \downarrow_R u_2$ for all critical pairs $u_1 = u_2$, then $\longrightarrow_R$ is locally confluent.*

For terminating unconditional HRS this yields a decision procedure for local confluence and hence for confluence.

Although we have not mentioned this aspect so far, it is obvious that the critical pair lemmas gives rise to a completion algorithm: critical pairs without common reduct are turned into rewrite rules and added to the non-confluent system. This process may need to be repeated to generate a confluent system. It is not clear whether the standard results on completion (e.g. [1]) carry over to higher-order systems.

5 λ^* and Other Applications

This section applies the critical pair lemma to various λ-calculi to prove that they are locally confluent. It follows that those systems are confluent for all terms which don't have infinite reductions. For typed systems which are terminating, this implies confluence for all well-typed terms.

The syntax of each system is defined by a signature which is a set of types

plus a set of typed constants. Terms are generated by variables and by application and abstraction, as defined in Section 2.1, subject to the type constraints in the signature. The convention of sticking to x, y and z for variables is relaxed now: and identifier which is not a declared constant can be used as a variable. In order to distinguish the defined equality between terms from the syntactic equality $=$, the former is denoted by $==$.

We start with some simple λ-calculi familiar from the literature.

The syntax of the pure λ-calculus involves just the type *term* of terms and two constants for application and abstraction:

$$\begin{aligned} \_.\_ &: term o term o term \\ \lambda &: (term o term) o term \end{aligned}$$

Assume that . associates to the left just like ordinary function application. Object-level equalities are

$$\begin{aligned} \beta: &\quad \lambda(f).s == f(s) \\ \eta: &\lambda([x]s.x) == s \end{aligned}$$

The rules β and η on their own do not generate any (non-trivial) critical pair. Their combination, however, gives rise to two critical pair which stem from the solutions to the two unification problems $\lambda(f) \equiv \lambda([x]s.x)$ and $[x]s.x \equiv [x]\lambda(g(x)).t(x)$:

$$
\begin{array}{ll}
 & s.t \\
 & \nearrow_\beta \\
\lambda([x]s.x).t & \\
 & \searrow_\eta \\
 & s.t
\end{array}
$$

$$
\begin{array}{ll}
 & \lambda([x]h(x)) \equiv \lambda(h) \\
 & \nearrow_\beta \\
\lambda([x]\lambda(h).x) & \\
 & \searrow_\eta \\
 & \lambda(h)
\end{array}
$$

Both critical pair are trivially joinable.

Now we consider the combination of λ-calculus and "algebraic", i.e. first-order, reductions as in [2]. The first-order term $f(s_1, \ldots, s_n)$ translates to the *term* $f.t_1 \ldots t_n$, where t_i is the translation of s_i. It is easy to see that the combination of β-reduction with the translation of algebraic term-rewriting systems does not generate any new critical pairs. In fact, we have the more general lemma:

Lemma 5.1 *Let R be a set of unconditional rules whose left-hand sides contain no λ and no subterm of the form $x.t$ where x is free. If R is locally confluent, so is $R \cup \{\beta\}$.*

Comparing this with Theorem 2.3 in [2] which states that if R is confluent, so is $R \cup \{\beta\}$, we find that the above lemma admits a larger class of rules R but requires termination of R to deduce confluence.

Our critical pair approach also explains why the addition of η to R may distroy confluence: new critical pairs may arise.

A well-known instance of R in Lemma 5.1 are the three rules for products with surjective pairing:

$$
\begin{array}{rl}
\pi_1 : & \pi_1\langle s, t\rangle == s \\
\pi_2 : & \pi_2\langle s, t\rangle == t \\
\pi : & \langle \pi_1 s, \pi_2 s\rangle == s
\end{array}
$$

where

$$
\begin{array}{rl}
\langle \_, \_ \rangle & : termotermoterm \\
\pi_1, \pi_2 & : termoterm
\end{array}
$$

Surjective pairing (π) creates two symmetric critical pairs with π_1 and pi_2, one of which we show:

$$
\begin{array}{ll}
 & \langle s, t\rangle \\
 & \nearrow_\pi \\
\langle \pi_1\langle s, t\rangle, \pi_2\langle s, t\rangle\rangle & \\
 & \searrow_{\pi_1}
\end{array}
$$

$$\langle s, \pi_2\langle s, t\rangle\rangle$$

The pair is easily joinable using the rule π_2. This yields that the untyped λ-calculus with surjective pairing is locally confluent. This is independent of whether we consider β, η or $\beta+\eta$ because none of the pairing rules overlap with β or η. Confluence holds only for terminating fragments, e.g. typable terms.

5.1 λ^*

The final and most complex application is a variation on the simply typed λ-calculus with products. The motivation for this calculus comes from an attempt to generalize Huet's unification algorithm for the simply typed λ-calculus [6] to a system with ML-style polymorphism documented in [11]. A complete solution to this problem seems to require backtracking over an infinite set of types $\alpha_1 o \cdots o \alpha_n o \tau$ which does not have a finite representation using simple types. In λ^* there is a product $*$ which is related to o by the equation $(\alpha * \beta) o \tau = \alpha o \beta o \tau$. In this system the types $\alpha_1 o \cdots o \alpha_n o \tau$ can be represented by the type $\alpha o \tau$, where α is a type variable. Replacing α by $\alpha_1 * \cdots * \alpha_n$ yields $\alpha_1 o \cdots o \alpha_n o \tau$.

There are the three types *type*, *var* and *term* and the following constants:

$$\begin{aligned}
1 &: \mathit{type} \\
_ * _, _ \Rightarrow _ &: \mathit{type} o \mathit{type} o \mathit{type} \\
_ \uparrow _ &: \mathit{var} o \mathit{type} o \mathit{term} \\
\langle\rangle &: \mathit{term} \\
\langle _, _ \rangle, _ . _ &: \mathit{term} o \mathit{term} o \mathit{term} \\
\lambda &: \mathit{type} o (\mathit{term} o \mathit{term}) o \mathit{term} \\
_ : _ &: \mathit{term} o \mathit{type} o \mathit{prop}
\end{aligned}$$

The type *prop* is the type of *propositions*. Note that ":" is only used as an object-level constant in this section. The letters α, β, γ, σ, and τ stand for object-level types. Products $(*)$ bind stronger than function spaces $(\Rightarrow)$. We write $\tau_1 * (\tau_2 * (\cdots * \tau_n)\ldots)$ as $\tau_1 * \cdots * \tau_n$ or even $\overline{\tau_n}$ (where $\overline{\tau_0} = 1$), and $\langle s_1, \langle s_2, \ldots, s_n\rangle \ldots\rangle$ as $\langle s_1, \ldots, s_n\rangle$ or even $\langle\overline{s_n}\rangle$ (where $\langle\overline{s_0}\rangle = \langle\rangle$ and $\langle\overline{s_1}\rangle = s_1$).

Reduction rules for types:

$$\begin{aligned}
(\alpha * \beta) * \gamma &== \alpha * (\beta * \gamma) && (107) \\
\alpha * \beta \Rightarrow \gamma &== \alpha \Rightarrow \beta \Rightarrow \gamma \\
\alpha \Rightarrow \beta * \gamma &== (\alpha \Rightarrow \beta) * (\alpha \Rightarrow \gamma) \\
\alpha * 1 &== \alpha && (108) \\
1 * \alpha &== \alpha && (109) \\
1 \Rightarrow \alpha &== \alpha \\
\alpha \Rightarrow 1 &== 1
\end{aligned}$$

The critical pair lemma for term rewriting systems [7] suffices to determine that the type reduction rules are locally confluent. Since they also terminate, they are confluent. If a type does not contain 1 or $*$, it is called *simple*. Thus the normal form of a type is a product of simple types (where the empty product is 1).

Type inference:

$$
\begin{aligned}
v^\tau &: \tau \\
\langle\rangle &: 1 \\
s:\sigma \;\wedge\; t:\tau &\Longrightarrow \langle s,t\rangle : \sigma * \tau \\
s:\sigma\Rightarrow\tau \;\wedge\; t:\sigma &\Longrightarrow s.t:\tau \qquad (110)\\
\bigwedge x.\ f(x^\sigma):\tau &\Longrightarrow \lambda(\sigma,f):\sigma\Rightarrow\tau
\end{aligned}
$$

(111)

The rules as they stand yield an obvious type inference algorithm but have a serious defect: they do not take the type equivalences into account. For example $pair.\langle\rangle$ does not have a type, although it should have type 1 because $\langle\rangle : 1$ and $1 == 1 \Rightarrow 1$. There are various ways in which $==$ can be built into type inference. The simplest solution consists of adding the rule

$$t:\sigma \;\wedge\; \sigma == \tau \quad\Longrightarrow\quad t:\tau$$

This destroys termination of the algorithmic interpretation of the rules. A more sophisticated solution is to work with $==$-equivalence classes of types. In logic programming parlance, this entails unification of types modulo $==$. Both solutions are logically equivalent and serve as the desired interpretation we attach to type inference in the sequel.

Unfortunately, this lands us with a difficult unification problem: it contains associative unification and may be a lot harder. However, if the class of queries is restricted to what we call *type inference problems*, i.e. those of the form $s : \alpha$, where α is a type variable and s does not contain any type variables, the above type inference rules behave fairly well if interpreted as a logic program: unifying $s : \alpha$ with the head of a clause simply requires the instantiation of α. In addition, we find that the predicates in the body of all clauses are again type inference problems, with the exception of rule (110). Hence we reformulate this rule as

$$s:\sigma \;\wedge\; t:\tau \;\wedge\; \sigma == \tau\Rightarrow\tau' \quad\Longrightarrow\quad s.t:\tau' \qquad (112)$$

which is logically equivalent to (110).

Now we have isolated the only non-trivial type unification problem, which is of a very special kind: when solving $\sigma == \tau \Rightarrow \tau'$ we can assume that τ' is a type variable and that σ and τ do not contain type variables. The latter is the consequence of a simple induction on the derivations which shows that all solutions to a problem $s : \alpha$, where α is a type variable and s does not contain any type

variables, are ground.

In order to solve an equation $\sigma == \tau \Rightarrow \alpha$ where σ and τ are ground and α is a variable, σ and τ are first put into normal form. This results in the equation $\overline{\sigma_m} == \overline{\tau_n} \Rightarrow \alpha$ whose normal form is $\overline{\sigma_m} == \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \alpha$. Therefore α must be of the form $\overline{\alpha_m}$, resulting in the set of independent equations

$$\sigma_i \;==\; \tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \alpha_i$$

which can be solved by unification in the empty theory. Hence there is at most one (modulo $==$) solution to the equation we started out with.

Thus we have shown that for type inference problems the modified rules can be used as an ordinary logic program (i.e. without fancy unification), except for the predicate $\sigma == \tau \Rightarrow \tau'$, which is treated separately. This yields a terminating algorithm which computes at most one type for each type inference problem, *the* type of the term.

Next we have the reduction rules for terms:

$$t : \tau \;\wedge\; \tau == \sigma \quad\Longrightarrow\quad \lambda(\sigma, f).t {==} f(t) \tag{113}$$
$$\langle\langle s, t\rangle, u\rangle {==} \langle s, \langle t, u\rangle\rangle \tag{114}$$
$$\langle\langle\rangle, t\rangle {==} t \tag{115}$$
$$\langle t, \langle\rangle\rangle {==} t \tag{116}$$
$$\lambda(\sigma, [x]\langle f(x), g(x)\rangle) {==} \langle\lambda(\sigma, f), \lambda(\sigma, g)\rangle \tag{117}$$
$$\lambda(\sigma, [x]\langle\rangle) {==} \langle\rangle \tag{118}$$
$$\langle s, t\rangle.u {==} \langle s.u, t.u\rangle \tag{119}$$
$$\langle\rangle.t {==} \langle\rangle \tag{120}$$
$$s.\langle t, u\rangle {==} (s.t).u \tag{121}$$
$$s.\langle\rangle {==} s \tag{122}$$
$$\lambda(\sigma * \tau, f) {==} \lambda(\sigma, [x]\lambda(\tau, [y]f(\langle x, y\rangle))) \tag{123}$$
$$\lambda(1, f) {==} f(\langle\rangle) \tag{124}$$
$$v^1 {==} \langle\rangle \tag{125}$$

A few remarks concerning rule (113) are in order.

- The precondition $t : \tau$ is decidable *if t does not contain type variables*, instantiating τ with a ground type, the type of t. Therefore $\tau == \sigma$ can be decided by normalization *if σ is a ground type*. Hence the rewrite relation on terms is decidable.
- In contrast to most other typed λ-calculi, run-time type checking is mandatory. The untyped version of (113), together with the other equalities, is inconsistent:

$$s == \langle\langle\rangle, s\rangle == \lambda(1 * \sigma, [x]x).\langle\langle\rangle, s\rangle == (\lambda(1 * \sigma, [x]x).\langle\rangle).s == \langle\rangle.s == \langle\rangle$$

- The proposition $t : \tau$ is closed under rewriting: all rewrite rules leave the type of a term invariant and type inference operates on equivalence classes of types.

We claim that this system is locally confluent on all terms without free variables of product type. More precisely, S is the set of terms whose β-normal form (meta, not object-level!) does not contain a subterm of the form v^τ where $\tau == \tau_1 * \tau_2$ for some $\tau_1, \tau_2 =/= 1$. It is easy to see that S is closed under $\equiv$, $\longrightarrow$, and taking subterms. Therefore Corollary 4.1 is applicable.

This system has a large number of critical pairs, most of which are directly joinable. There are, however, some complicated cases which we concentrate on. Their treatment requires an important lemma.

Lemma 5.2 *If $s \in S$ and $s : \sigma$, where $\sigma = \sigma_1 * \cdots * \sigma_n$ is in normal form w.r.t. type reduction, then $s \longrightarrow^* \langle s_1, \ldots, s_n\rangle$ where $s_i : \sigma_i$.*

Now we examine all overlaps of object-level β-reduction with other rules which yield the only nontrivial critical pairs.

Overlapping (113) and (117) generates:

$$\begin{array}{lll} & & \langle f(t), g(t)\rangle \\ & & \nearrow_{113} \\ t : \sigma & \Longrightarrow \lambda(\sigma, [x]\langle f(x), g(x)\rangle).t & \\ & & \searrow_{117} \\ & & \langle \lambda(\sigma, f), \lambda(\sigma, g)\rangle.t \end{array}$$

Because $t : \sigma$ we can join the pair again: $\langle \lambda(\sigma, f), \lambda(\sigma, g)\rangle.t \longrightarrow_{119} \langle \lambda(\sigma, f).t, \lambda(\sigma, g).t\rangle \longrightarrow^2_{113} \langle f(t), g(t)\rangle$.

Overlapping (113) and (118) generates:

$$\begin{array}{lll} & & \langle\rangle \\ & & \nearrow_{113} \\ t : \sigma & \Longrightarrow \lambda(\sigma, [x]\langle\rangle).t & \\ & & \searrow_{118} \\ & & \langle\rangle.t \end{array}$$

and we immediately have $\langle\rangle.t \longrightarrow_{120} \langle\rangle$.

Overlapping (113) and (121) generates:

$$\begin{array}{lll} & & f(\langle s, t\rangle) \\ & & \nearrow_{113} \\ \langle s, t\rangle : \gamma & \Longrightarrow \lambda(\gamma, f).\langle s, t\rangle & \\ & & \searrow_{121} \\ & & (\lambda(\gamma, f).s).t \end{array}$$

Joining this critical pair is slightly more involved. Since $\langle s, t\rangle : \gamma$, there are σ_i such that $s_i : \sigma_i$ and $\sigma * \tau == \gamma$. Let $\overline{\sigma_j}$ and $\overline{\tau_k}$ be the normal forms of σ and τ respectively. Appealing to Lemma 5.2 there are $s_j : \sigma_j$ and $t_k : \tau_k$ such that $s \longrightarrow^* \langle\overline{s_j}\rangle$ and $t \longrightarrow^* \langle\overline{t_k}\rangle$. Hence $f(\langle s, t\rangle) \longrightarrow^* f(\langle\langle\overline{s_j}\rangle, \langle\overline{t_k}\rangle\rangle) \longrightarrow^* f(\langle\overline{s_j}, \overline{t_k}\rangle)$ and

$$\begin{array}{l} (\lambda(\gamma, f).s).t \;\longrightarrow^*\; (\lambda(\gamma, f).\langle\overline{s_j}\rangle).\langle\overline{t_k}\rangle \;\longrightarrow^*\; (\lambda(\overline{\sigma_j} * \overline{\tau_k}, f).\langle\overline{s_j}\rangle).\langle\overline{t_k}\rangle \\ \longrightarrow^* (\lambda(\sigma_1, [x_1] \ldots \lambda(\sigma_j, [x_j]\lambda(\tau_1, [y_1] \ldots \lambda(\tau_k, [y_k]f(\langle\overline{s_j}, \overline{t_k}\rangle)) \ldots)) \ldots).\langle\overline{s_j}\rangle).\langle\overline{t_k}\rangle \\ \longrightarrow^* f(\langle\overline{s_j}, \overline{t_k}\rangle) \end{array}$$

Overlapping (113) and (122) generates:

$$
\begin{array}{lcl}
 & & f(\langle\rangle) \\
 & \nearrow_{113} & \\
\langle\rangle : \sigma \;\Longrightarrow\; \lambda(\sigma, f).\langle\rangle & & \\
 & \searrow_{122} & \\
 & & \lambda(\sigma, f)
\end{array}
$$

From $\langle\rangle : \sigma$ it follows that $\sigma == 1$ and thus $\sigma \longrightarrow^* 1$. Thus $\lambda(\sigma, f) \longrightarrow^* \lambda(1, f) \longrightarrow_{124} f(\langle\rangle)$.

Overlapping (113) and (123) generates:

$$
\begin{array}{lcl}
 & & f(t) \\
 & \nearrow_{113} & \\
t : \sigma * \tau \;\Longrightarrow\; \lambda(\sigma * \tau, f).t & & \\
 & \searrow_{124} & \\
 & & \lambda(\sigma, [x]\lambda(\tau, [y]f(\langle x, y\rangle))).t
\end{array}
$$

Joining this critical pair is very similar to the case of overlapping (113) and (121) and is left to the interested reader.

Overlapping (113) and (124) generates:

$$
\begin{array}{lcl}
 & & f(t) \\
 & \nearrow_{113} & \\
t : 1 \;\Longrightarrow\; \lambda(1, f).t & & \\
 & \searrow_{124} & \\
 & & f(\langle\rangle).t
\end{array}
$$

Since $t : 1$, we have $t \longrightarrow^* \langle\rangle$ and thus $f(t) \longrightarrow^* f(\langle\rangle)$ and $f(\langle\rangle).t \longrightarrow^* f(\langle\rangle).\langle\rangle \longrightarrow_{122} f(\langle\rangle)$.

For completeness we record the remaining critical pairs which are easily joinable:

$$\begin{array}{lc}
(114)+(115): & \langle\langle\rangle,\langle t,u\rangle\rangle{=}\langle t,u\rangle \\
(114)+(116): & \langle s,\langle\langle\rangle,u\rangle\rangle{=}\langle s,u\rangle \\
(114)+(116): & \langle s,\langle t,\langle\rangle\rangle\rangle{=}\langle s,t\rangle \\
(114)+(117): & \lambda(\sigma,[x]\langle f(x),\langle g(x),h(x)\rangle\rangle){=}\langle\lambda(\sigma,[x]\langle f(x),g(x)\rangle),\lambda(\sigma,h)\rangle \\
(114)+(119): & \langle s,\langle t,u\rangle\rangle.r{=}\langle\langle s,t\rangle.r,u.r\rangle \\
(114)+(121): & (r.\langle s,t\rangle).u{=}r.\langle s,\langle t,u\rangle\rangle \\
(115)+(116): & \langle\rangle{=}\langle\rangle \\
(115)+(117): & \lambda(\sigma,g){=}\langle\lambda(\sigma,[x]\langle\rangle),\lambda(\sigma,g)\rangle \\
(115)+(119): & t.u{=}\langle\langle\rangle.u,t.u\rangle \\
(116)+(117): & \lambda(\sigma,f){=}\langle\lambda(\sigma,f),\lambda(\sigma,[x]\langle\rangle)\rangle \\
(116)+(119): & s.u{=}\langle s.u,\langle\rangle.u\rangle \\
(117)+(123): & \langle\lambda(\sigma*\tau,f),\lambda(\sigma*\tau,g)\rangle{=}\lambda(\sigma,[x]\lambda(\tau,[y]\langle f(\langle x,y\rangle),g(\langle x,y\rangle)\rangle)) \\
(117)+(124): & \langle\lambda(1,f),\lambda(1,g)\rangle{=}\langle f(\langle\rangle),g(\langle\rangle)\rangle \\
(118)+(123): & \langle\rangle{=}\lambda(\sigma,[x]\lambda(\tau,[y]\langle\rangle)) \\
(118)+(124): & \langle\rangle{=}\langle\rangle \\
(119)+(121): & \langle s.\langle u_1,u_2\rangle,t.\langle u_1,u_2\rangle\rangle{=}(\langle s,t\rangle.u_1).u_2 \\
(119)+(122): & \langle s.\langle\rangle,t.\langle\rangle\rangle{=}\langle s,t\rangle \\
(120)+(121): & \langle\rangle{=}(\langle\rangle.t).u \\
(120)+(122): & \langle\rangle{=}\langle\rangle \\
(123)+(107): & \lambda(\alpha*\beta,[x]\lambda(\gamma,[y]f(\langle x,y\rangle))){=}\lambda(\alpha*(\beta*\gamma),f) \\
(123)+(108): & \lambda(1,[x]\lambda(\alpha,[y]f(\langle x,y\rangle))){=}\lambda(\alpha,f) \\
(123)+(109): & \lambda(\alpha,[x]\lambda(1,[y]f(\langle x,y\rangle))){=}\lambda(\alpha,f)
\end{array}$$

We have left out any critical pairs arising from overlapping two type reduction rules because of their first-order nature.

Adding the η-rule

$$\lambda([x]s.x) \ == \ s \tag{126}$$

yields the two trivial critical pairs with β-reduction and the following nontrivial ones with the remaining rules:

$$\begin{array}{ll}
(126)+(119): & \langle s,t\rangle{=}\lambda(\sigma,[x]\langle s.x,t.x\rangle) \\
(126)+(120): & \langle\rangle{=}\lambda(\sigma,[x]\langle\rangle) \\
(126)+(123): & s{=}\lambda(\sigma,[x]\lambda(\tau,[y]s.\langle x,y\rangle)) \\
(126)+(124): & s{=}s.\langle\rangle
\end{array}$$

All of them are joinable.

All of this yields the theorem

Theorem 5.1 *The system λ^* is locally confluent for all terms in S (with and without η-reduction).*

Furthermore, it is easy to see that λ^* is normalizing: eliminating $\langle\rangle$ and pushing $\langle\_,\_\rangle$ to the outside results in a tuple of simply typed λ-terms which are known to be normalizing. We strongly conjecture that λ^* is in fact terminating.

Finally we would like to point out that λ^* is closely connected with work in functional programming

[15, 16]. In the presence of pairs there is the problem of remembering whether a function was curried or not, i.e. had type $\alpha o \beta o \gamma$ or $\alpha * \beta o \gamma$. λ^* is a language where this distinction has become immaterial. Note, however, that the order of arguments still matters.

Acknowledgements

The author wishes to thank Eugenio Moggi and Andrew Pitts for many lengthy discussions on λ^* and related topics.

References

[1] L. Bachmair: *Canonical Equational Proofs*, Research Notes in Theoretical Computer Science, Wiley and Sons (1989).

[2] V. Breazu-Tannen: *Combining Algebra and Higher-Order Types*, Proc. IEEE Symp. Logic in Computer Science (1988).

[3] N. Dershowitz, J.-P. Jouannaud: *Rewrite Systems*, in J. van Leeuwen (editor), Handbook of Theoretical Computer Science, Vol B: Formal Methods and Semantics, North-Holland, to appear.

[4] M. Gordon: *HOL, A Machine Oriented Formulation of Higher Order Logic*, Tech. Rep. 68, Computer Lab., Univ. of Cambridge (1985).

[5] J.R. Hindley, J.P. Seldin: *Introduction to Combinators and λ-Calculus*, Cambridge University Press (1986).

[6] G.P. Huet: *A Unification Algorithm for Typed λ-Calculus*, Theoretical Computer Science 1 (1975), 27-57.

[7] G. Huet: *Confluent Reductions: Abstract properties and Applications to Term Rewriting Systems*, J. ACM 27, 4 (1980), 797-821.

[8] J.W. Klop: *Combinatory Reduction Systems*, Ph.D. Thesis, Univ. Utrecht; publ. Mathematisch Centrum, Amsterdam (1980).

[9] D.E. Knuth, P.B. Bendix: *Simple Word Problems in Universal Algebras*, in: Computational Problems in Abstract Algebra, ed J. Leech, Pergamon Press (1970), 263-297.

[10] D. Miller: *A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification*, to appear in *Extensions of Logic Programming*, LNCS, Springer-Verlag.

[11] T. Nipkow: *Higher-Order Unification, Polymorphism, and Subsorts*, Proc. 2nd Intl. Workshop on Conditional and Typed Rewriting Systems, Montreal, June 1990, to appear in LNCS.

[12] B. Nordström: *Martin-Löf's Type Theory as a Programming Logic*, Report 27, PMG, Chalmers Univ. of Technology, Dept. of Comp. Sci. (1986).

[13] L.C. Paulson: *Isabelle: The Next 700 Theorem Provers*, in P. Odifreddi (editor), Logic and Computer Science, Academic Press (1990), in press, 361-385.

[14] L. C. Paulson: *The Foundation of a Generic Theorem Prover*, Journal of Automated Reasoning 5 (1989), 363–397.

[15] M. Rittri: *Using Types as Search Keys in Function Libraries*, Proc. 4th Intl. Conf. Functional Programming Languages and Computer Architecture (1989), 174-183.

[16] M. Rittri: *Retrieving Library Identifiers whose Types are at Least as General, Modulo CCC-Isomorphisms, as a Given Type*, Proc. 10th Intl. Conf. on Automated Deduction (1990), LNCS ???.

[17] W. Snyder, J. Gallier: *Higher-Order Unification Revisited: Complete Sets of Transformations*, J. Symbolic Computation (1989) 8, 101-140.

Extracting and Executing Programs Developed in the Inductive Constructions System: a Progress Report

Christine Paulin & Benjamin Werner

CNRS/ENS Lyon - INRIA Rocquencourt

Abstract

The experiment we describe here, consists in using program specifications proved in the Calculus of Constructions enhanced by inductive definitions, to extract their computational contents, and to execute them in a high-level functional language. We try to explain the need for a well-adapted target language to execute the mechanically extracted programs. In order to point out the related problems, we realized a first rudimentary implementation and we show the system working on an classical example.

1 Introduction

1.1 Brief review

The Calculus of Constructions, is now a well understood formal logical system. The implementation realized at INRIA Rocquencourt has been distributed to 32 sites (for an exact description, see [9]). Let us just recall that it is a higher-order λ-calculus, which can be seen as F_ω extended with dependent types. This gives a very convenient framework for developing mechanically checked constructive proofs (see [3], [4], [6], [8], etc). This feature can be used in order to write programs which are certified to terminate and meet a logical specification: if we prove an existencial formula, like $\forall x : X\ \exists y : Y.(\mathit{Specif}(x,y))$, then given any value x_0 of type X, we can use the proof to compute a value y_0 of type Y, which will be certified to meet the specification Specif.

The extraction algorithm of C. Paulin (see [13] and [14]) describes how to get rid of the part of the logical specification with no computational contents. In the previous example, we would just keep the method to compute y_0, without worrying about the proof of $\mathit{Specif}(x_0, y_0)$. By doing this, we eliminate the dependent types; that means that the programs we extract are terms of the F_ω higher-order λ-calculus. If we use the version of the Calculus of Constructions extended by inductive types, then we extract in F_ω extended by inductive types, also called F_ω^{idt} by Pfenning and Michaylov in [12].

The resulting program consists in a finite sequence of definitions, each of them comprising an identifier and a λ-term (or possibly an inductive type definition, a type constructor or elimination). The problem is then to understand how to execute such a program. In the following, we will try to describe this last step.

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks" and by the Programme de Recherches Coordonnées "Programmation Avancée et Outils pour l'Intelligence Artificielle".

1.2 Motivations

Our final goal is the realization of an environment for the development of certified correct programs. This system should be convenient enough for the user and reasonably efficient in execution. In order to achieve this, we believe it is rational to extract from the logical level to a high-level programming language for the following reasons:

- If we want someday to produce acceptable programs, we will have anyway to use standard features of programming languages, such as binary integers, modularity, exceptions, etc.
- If the "programming style" of the extracted programs is too exotic to run efficiently on a programming language, then it will probably be difficult to make it run efficiently at all. Therefore it is preferable to spend time on trying to obtain more rational programs, than trying to compile them unmodified.
- In many cases, the programmer will not want to specify the whole software system, but just a set of routines. Extracting in a usable language allows us to use those specified routines, just as one would do if there were "made by hand". Note that this raises the interesting module design problem, of linking the modules of the logical level with the ones of the execution language.
- The compilation step should be left to the compilation specialists working on programming languages.

2 The Target Language

2.1 Definition

Studying the problems related to the execution of extracted programs means essentially to define a well-adapted language for these programs. It is clear that we cannot give any precise specification yet; though it appears that we will have to use three main tools:

- The λ-calculus, with usual abstraction and application.
- Inductively defined positive types and primitive recursion over them.
- Some built-in efficient structures such as binary integers and usual operations over them.

We will call this embryonal language "FML".

It is certainly an important point to define a well-adapted type system for FML. However, at this point, we have not carefully studied this problem, and thus the user may think of FML programs as being either untyped or else typed with ML-like types.

We use the following notation for extracted terms:

- $[x]t$ for abstraction over x.

- $t_1 \ t_2$ for application.
- $Constr\{T, n\}(t_1, t_2, \ ... \ , t_m)$ for the n^{th} constructor of a given inductive type T, applied to its m arguments t_i.
- $Elim \ \{T\}(f_1, f_2, \ ... \ , f_n)$ for the elimination scheme over an inductive type T (see 2.2).

2.2 Inductive types and primitive recursion

The theoretical framework is described in [7]. Let us just recall the essential points: each inductive type T is described by the list of its constructors; each constructor $Constr\{T, k\}$ is described by its type Ψ_k which is of the form:

$\Psi_k = \Phi_1 \to \Phi_2 \to ... \Phi_n \to T$

where T does not appear in Φ_i, or is strictly positive, i.e.:

$\Phi_i = T_i^1 \to T_i^2 \to ... \to T_i^{\ m_i} \to T.$

with T not occuring in T_i^j

An object of type T, built with the k^{th} constructor will then be of the form:

$Constr\{T, k\}(A_1, A_2, \ ... \ , A_{n_k}).$

The elimination scheme associated to type T, will be defined by the following reduction rule:

$Elim\{T\}(f_1, f_2, \ ... \ , f_m \ Constr\{T, k\}(A_1, A_2, \ ... \ , A_n)) \ \triangleright$

$(f_k \ A_1 \ A_2 \ ... \ A_n \ (\phi_{j_1} \ A_{j_1}) \ (\phi_{j_2} \ A_{j_2}) \ \ ... \ (\phi_{j_l} \ A_{j_l})).$

where m is the number of constructors of T and the j_p's are the indexes of the Φ_i's where T occurs. ϕ_i being defined by:

$\phi_i \equiv [t][x_1][x_2]...[x_{mi}]Elim\{T\} \ (f_1, f_2, \ ..., \ f_m)(t \ x_1 \ x_2 \ ... \ x_{mi}).$

2.3 Translation in ML

As explained above, the idea is to execute the extracted programs in a high-level language which can also directly be used by the programmer. As all the FML features are present and easy to use in ML, it seems natural to use this language to make the extracted programs run. In this very first prototype, we used LML, which is a lazy implementation of ML, developed by L. Augustsson and Th. Johnsson in Göteborg. We used it because at this point we till had some little problems with the strict evaluation mechanism of other ML compilers. This last point is explained in

Let us now see how the features we have described can be encoded in ML using concrete types and simple pattern matching. Before treating a more substantial problem, we will take a look at a simple example.

The classical structure of lists is defined as follows in the constructions:

```
Inductive Definition list [A:Data] : Data =
```

```
  nil : (list A)
| cons : A -> (list A) -> (list A).
```

The system then builds:

- the *list* type of type $Data \rightarrow Data$
- the constructors *nil* and *cons*, respectively of type

  $(A : Data)(List\ A)$

  and

  $(A : Data)A \rightarrow (List\ A) \rightarrow (List\ A)$.

  As in most functional languages, *nil* stands for the empty list, and *cons* for the function building the list defined by its first element and the rest of the list.
- a simple elimination function *list_recd* of type

  $(A : Data)(P : Data)P \rightarrow (A \rightarrow (list\ A) \rightarrow P \rightarrow P) \rightarrow (list\ A) \rightarrow P$.

  The third argument of the function, which is of type P, will be the returned if the list over which we do the elimination (i.e. the last argument) is *nil*; if we make the elimination over $(cons\ a\ l)$, the result will be the fourth argument applied to a, l and the image of l by the elimination.
- functions providing dependent elimination, resp. called *list_rec*, *list_ind*, and *list_recs*, of type

  $(A : Data)$

  $(P : (list\ A) \rightarrow \kappa)$

  $(P\ (nil\ A)) \rightarrow ((a : A)(l0 : (list\ A))(P\ a) \rightarrow (P\ (cons\ A\ a\ l0))) \rightarrow (l : list\ A)(P\ l)$

  where κ resp. stands for *Type*, *Prop* or *Spec*. This function is similar to the previous one, but allows to prove properties by induction over the lists structure.

We now can define the *append* function over lists, which returns the concatenation of its two arguments. In CAML it would be written:

```
#let rec append l1 l2 = match l1 with
#  [] -> l2
#| (a::l) -> (a::(append l l2));;

Value append = <fun>:('a list -> 'a list -> 'a list)
```

In the Calculus of Constructions, the pattern matching is replaced by the elimination scheme:

```
Definition append.
Body
[A:Data][l1:(list A)][l2:(list A)]
(list_recd A (list A) l2 [a:A][l:(list A)][l3:(list A)](cons A a l3) l1).
```

Here are the FML terms extracted from the previous inputs:

```
list[A] == nil+cons:A (list A).

nil == Constr{list,1}
cons == [Var1][Var2]Constr{list,2}(Var1,Var2)
list_recs == [Var1][Var2][Var3]Elim{list}(Var1,Var2,Var3)
list_recd == [Var1][Var2][Var3]Elim{list}(Var1,Var2,Var3)
append == [l1][l2](list_recd l2 [a][l][l3](cons a l3) l1)
```

And finally the translation in LML:

```
let type list *A  = nil  + cons  *A (list *A) in
let rec list_elim =   \f_x1.  \f_x2. \C.( case C in
   (nil) : f_x1
|| (cons a_x1 a_x2) : f_x2 a_x1 a_x2 (list_elim f_x1 f_x2 a_x2) end)
in
let list_recs = \Var1.\Var2.\Var3.(list_elim Var1 Var2 Var3)
in
let list_recd = \Var1.\Var2.\Var3.(list_elim Var1 Var2 Var3)
in
let append = \l1.\l2.(list_recd l2 (\a.\l.\l3.(cons a l3)) l1)
in
...
```

3 An Example: Running a sorting algorithm

3.1 The quicksort algorithm and its specification

Quicksort is a well-known algorithm. It is defined by recursion on the length of the list to be sorted: We want to sort a list $[a_1, a_2, a_3, \ \ldots\ , a_n]$, of length n, relatively to a given pre-ordering $<$.

- if $n = 0$, i.e. if the list is empty, it is trivially sorted.
- if $n \geq 1$, we consider the first element a_1. Then we define two new lists l and l', respectively comprising the elements of $[a_2, a_3, \ \ldots\ , a_n]$ which are greater (resp. not greater) than a_1. The length of each of these two lists being less than n, we can sort them by the induction hypothesis, getting two sorted lists l_1 and l'_1.

  We then get the wanted result, by concatenating the three lists l_1, $[a_0]$ and l'_1.

Here is a CAML program example:

```
#let Splitting inf_sup x l = Split x [] [] l
#        where rec
```

```
#    Split x l1 l2 = function
#         [] -> (l1,l2)
#        |(a::l) -> if (inf_sup a x) then (Split x l1 (a::l2) l)
#                                    else (Split x (a::l1) l2 l)

Value Splitting = <fun> : (('a -> 'a -> bool)-> 'a -> 'a list -> 'a list * 'a list)

#let sort less = quicksort where
#let rec quicksort  = function
#    [] -> []   (* the empty list is sorted *)
#   |a::l -> let (l1,l2) = (Splitting less a l) in
#             (quicksort l2)@(a::(quicksort l1))
#;;

Value sort = <fun> : (('a -> 'a -> bool) -> 'a list -> 'a  list)
```

One can remark that this is not the very real quicksort algorithm, as we use the append function, which gives us an execution time in $O(n^2)$. However, this simplification, which could in fact be avoided, is not fundamentely relevant for the part of the problem we are here interested in.

Proving the correctness of this algorithm is not difficult from an intuitive point of view. Three steps are not entirely trivial, when the proof is written formally:

- We have to specify how to split a list into two parts; one containing the elements greater than a given parameter, the other the elements not greater than it (in the formalized proof, this theorem is called *Splitting*).
- We have to prove the induction step; i.e. given two sorted lists l_1 and l_2 and given a variable a, if all elements of l_1 are less than a, if all elements of l_2 are greater than a, then the list $l_1@(a::l_2)$ is sorted, and contains the same elements as the list we started from.
- Finally we have to prove the induction principle allowing us to reason explicitely on the length of the lists (this corresponds to the theorem called *induction* in the formalized proof).

3.2 The specification in the Calculus of Constructions and its realization

Once the specification is proved, we can get rid of the dependent types and the proof parts with no computational content by extracting in F_ω^{idt}. As giving the whole proof of the program specification would be too long, let us focus on a single point, which is significant enough, to show the mechanism working.

As it can be seen above, the definition of the quicksort algorithm, is a recursion based on the length of the list, and not an induction based on the structure of the list. Therefore, the elimination scheme described in cannot be used directly, and we have to prove some specific recursion theorem, which can be stated as the following:

Theorem: Considering a predicate P over lists, such as:

- (P nil) is true
- given an element a and a list l_1, if every list l_2 of length not greater than that of l_1 satisfies (P l_2), then we have (P ($cons$ a l_1)).

Then for all lists l, we have (P l) is true.

In the Calculus of constructions, the length order relation over lists, named *lel* is defined by:

```
Definition length.
Body (list_recd nat O [a:A][m:list]S) : list->nat.

Definition lel [l,m:list](le (length l) (length m)).
(* le is the ordering over type nat *)
```

We then can state the theorem by:

```
(l:list)
(P:list->Spec)
(P nil)->
((a:A)(l1:list)((l2:list)(lel l2 l1)->(P l2))->(P (cons a l1)))->
(P l)
```

Our main tools for the proof, are of course the elimination schemes over *list* and some properties over *lel*. Here is how the proof is developed interactively using the tactics of the system. The user-entered commands are prompted by "$CoC->$".

We begin to state what we want to prove:

```
CoC ->Goal (l:list)(P:list->Spec)
     (P nil)->((a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons am)))
              ->(P l).
```

We then can separate the final goal from the hypothesises:

```
CoC ->Do intros.
```

The system returns:

```
1 subgoal
  (P l)
  ============================
    H0 : (a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m))
    H : (P nil)
    P : list->Spec
    l : list
```

As we make a proof on induction on the length of the list l, we substitute the goal with the stronger assertion *"P is verified for all lists shorter than l"*:

```
CoC ->Do cut (n:list)(lel n l)->(P n) THEN automatic.
```

We get:

```
1 subgoal
  (n:list)(lel n l)->(P n)
  ============================
    H0 : (a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m))
    H : (P nil)
    P : list->Spec
    l : list
```

We now use structural induction over l.

```
CoC ->Do elim l.

2 subgoals
  (n:list)(lel n nil)->(P n)
  ============================
    H0 : (a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m))
    H : (P nil)
    P : list->Spec
    l : list
subgoal 2 is:
CoC ->  (y:A)(y0:list)((n:list)(lel n y0)->(P n))->(n:list)(lel n (cons y y0))->(P n)
```

We see that the first subgoal stands for the case $l = nil$; the second one for $l = (cons\ y\ y0)$. The first case can easily be resolved, as it implies $n = nil$:

```
CoC ->Do intros THEN resolvew (eq_spec list nil n) THEN automatic.

1 subgoal
  (y:A)(y0:list)((n:list)(lel n y0)->(P n))->(n:list)(lel n (cons y y0))->(P n)
  ============================
    H0 : (a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m))
    H : (P nil)
    P : list->Spec
    l : list
```

We now make a proof by induction on n. The case $n = nil$ is trivial:

```
CoC ->Do intros_with a m H1 n THEN elim n THEN automatic.

1 subgoal
  (y:A)(y0:list)((lel y0 (cons a m))->(P y0))->
                    (lel (cons y y0) (cons a m))->(P (cons y y0))
```

For the case $n = (cons\ y\ y0)$ we use the $H0$ induction hypothesis.

```
CoC ->Do intros_with b p H2 H3 THEN resolve H0 THEN intros.
1 subgoal
  (P n0)
  ============================
    H4 : (lel n0 p)
    n0 : list
    H3 : (lel (cons b p) (cons a m))
    H2 : (lel p (cons a m))->(P p)
    p : list
    b : A
    n : list
    H1 : (n:list)(lel n m)->(P n)
    m : list
    a : A
    H0 : (a:A)(m:list)((n:list)(lel n m)->(P n))->(P (cons a m))
    H : (P nil)
    P : list->Spec
    l : list
```

At this point we have to prove $(P\ n0)$, for all $n0$ such as $(lel\ n0\ p)$. This is done easily, because $n0$ is shorter than p, and p is shorter than m because $(lel\ (cons\ b\ p)\ (cons\ a\ m))$.

If we dare looking at the proof term, here is what we see:

```
[A:Data]
 [l:(list A)]
  [P:(list A)->Spec]
   [H:(P (nil A))]
    [H0:(a:A)(m:(list A))((n:(list A))(lel A n m)->(P n))->(P (cons A a m))]
     (list_recs A [y:(list A)](n:(list A))(lel A n y)->(P n)
        [n:(list A)]
         [H1:(lel A n (nil A))]
          (eq_spec (list A) (nil A) n (lel_nil A n H1)
             [y:(list A)](P y) H)
        [a:A]
         [m:(list A)]
          [H1:(n:(list A))(lel A n m)->(P n)]
           [n:(list A)]
            (list_recs A [y:(list A)](lel A y (cons A a m))->(P y)
               [H2:(lel A (nil A) (cons A a m))]H
               [b:A]
                [p:(list A)]
                 [H2:(lel A p (cons A a m))->(P p)]
                  [H3:(lel A (cons A b p) (cons A a m))]
                   (H0 b p
```

```
                        [n0:(list A)]
                         [H4:(lel A n0 p)]
                          (H1 n0
                             (lel_trans A n0 p m H4
                                (lel_tail A b a p m H3))))
                n)
         l l (lel_refl A l))
      : (A:Data)
         (l:(list A))
          (P:(list A)->Spec)
           (P (nil A))->
            ((a:A)
              (m:(list A))
               ((n:(list A))(lel A n m)->(P n))->(P (cons A a m)))->(P l)
```

If we extract the computational contents of this big proof term, we get the following FML term:

```
induction ==
  [l][H][H0](list_recs [n]
                          (eq_spec nil n H)
                          [a][m]
                              [H1]
                               [n](list_recs H [b][p][H2](H0 b p [n0](H1 n0)) n)
                       l l)
```

Before generating the executable file, it is also possible to realize "by hand" some axioms used in the Specification, without having been proved. In the same way, it is possible to instanciate a type variable used at the logical level; this is a first (small) step towards the use of abstract types.

Once the extraction is done, we can use the specified *Quicksort* function. As the F_ω term began with a type abstraction, the obtained function is polymorphic; it can be applied to a list of build-in integers. That for we must use FML features to define a comparison function:

```
let inf_sup = [x,y] if x<y then left else right.

let l = (cons 3 (cons 1 (cons 6 (cons 10 (cons 4 nil))))).
```

It then is possible to generate a LML source file, which will execute any given FML statement:

```
FML -> Write File (Quicksort inf_sup l).
```

The execution of the obtained LML file will give us the expected result.

```
time sort -a
1 3 4 6 10
0.0u 0.0s 0:00 50% 0+21k 0+0io 0pf+0w
```

4 Discussion and open problems

4.1 Typing Problems

Running our program on a ML compiler, we did not first verify that we only had obtained extracted terms that were typable in the ML type system, which is not supposed to always be the case. In fact, it is very rare to extract a term that is too polymorphic for the ML type-system. And even in this last case, assuming the correctness of ML-compilers with respect to the evaluation of pure λ-calculus plus inductive types, we can ensure that the extracted programs will meet their specifications. Of course a future system will have to handle this problem more rigorously; yet, by now, we can imagine different solutions:

- Use an ML compiler that would skip the typechecking phase (that is possible in LML for instance). This is quite easy but not very satisfying.
- Detecting at the logical level all the terms, which would generate a non-typable ML function.
- Use a language with an extended more powerful type system. Such improvements are proposed by various research directions: F_ω typing style in QUEST [2] and LEAP [15], or various ML type system extensions based on Coppo's conjunctive types. Realizing a nice link between the type system of the logical level and the type system of the execution level could possibly be the most promising solution.

4.2 Evaluation modes

In this very first prototype, we used LML to execute the extracted programs. However, lazyness is not an obligation; as F_ω and F_ω^{idt} are strongly normalizing, the programs will also terminate in a strict evaluation mode, and give the same result. The only problem is that, in some cases, strictness causes some terms to be evaluated, that are not needed in the further computation. This is particularly harmful when the evaluated term is a recursive call; in such a case, the execution time can explode. However there should be no excessive difficulty to make extracted programs run correctly in CAML or some other strict version of ML. This could be achieved by rationalizing the elimination schemes, and using a certain amount of partial evaluation.

More generally, we want to define an operational semantics for FML. This semantics will have to insure the termination in accordance with the specifications. The translation into ML must then respect the programs computational behaviour.

4.3 Performances of the Extracted Programs

The LML source file we generate, is not very fancy and in some parts a little redundant. However it is readable for someone having understood the specification proof. If we look at the execution time, the extracted program is slower than an equivalent "hand-made" one, by a constant ratio of approximately 1.5 in the present case. One can make similar observations, looking at other examples, such as euclidian division or Warshall's algorithm for calculating the transitive closure of a given relation.

It seems that in some cases it could be useful to perform a certain amount of β-reductions before executing the program itself (partial evaluation). A good algorithm for performing this would certainly be a real improvement.

It remains unclear, whether or not, in certain cases, primitive recursion is sufficient to implement efficiently some algorithms and if some more general recursion forms could be useful.

A positive point is the mixing of extracted routines, with built-in data structures, that is made possible by the polymorphism of the extracted programs.

4.4 Conclusion

The experiment described in this note has mainly shown us that it can be realistic to extract from the logical level to a high-level language, and that ML offers a good framework for that approach. For the future, we can see three main points to look at:

- Defining a well-adapted type system for the extracted programs.
- See if it is useful to use more general forms of recursion, and how to encode them at the logical level.
- Implementing efficient programming tools, and encoding them at the logical level.

Should these goals be reached, and on the other side a performing logical toplevel based on tactics and evoluated mathematical vernacular be developed, we could imagine a realistic system for developing certified programs.

References

[1] L. Augustsson, Th. Johnsson. "Lazy ML user's manual." Preliminary Draft, Göteborg University, July 1989.

[2] L. Cardelli. "Typeful programming". Lecture Notes for the IFIP Advanced Seminar on Formal Methods in Programming Language Semantics. Rio de Janeiro, Brazil, 1989. SRC report #45, DEC, 1989.

[3] Th. Coquand. "Une Théorie des Constructions." Thèse de troisième cycle, Université Paris VII, 1985.

[4] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics", EUROCAL85, Linz, Springer-Verlag LNCS 203, 1985.

[5] Th. Coquand, G. Huet. "The Calculus of Constructions." Information and Computation, Volume 76, 1988.

[6] Th. Coquand. "Metamathematical investigations of a Calculus of Constructions." LCS edited by P. Oddifredi, Academic Press, 1990. also in [9] and in Rapport de recherche INRIA 1088, Sept. 89.

[7] Th. Coquand, Ch. Paulin. "Inductively Defined Types." Proc. of Colog 88, Ed. P. Martin-Löf & G. Mints, LCS 417, 1990.

[8] G. Dowek. "A Vernacular Syllabus." in [9]

[9] G. Huet Ed. "The Calculus of Constructions, Documentation and user's guide." Rapport technique n° 110, INRIA, 1989.

[10] G. Huet. "A Uniform Approach to Type Theory." Rapport de recherche INRIA n° 795 , Février 88, also in "Logical Foundation of Functional Programming", Addison-Wesley, 1990.

[11] J.-L. Krivine, M. Parigot. "Programming with Proofs." Preprint presented at 6th symposium on Computation Theory, Wendisch-Rietz, German Dem. Rep. 1987.

[12] S. Michaylov, F. Pfenning. "Compiling the Polymorphic λ-Calculus." *Extended Abstract.* Prepint. CMU, Pittsburgh, USA.

[13] Ch. Paulin-Mohring. "Extraction de programmes dans le Calcul des Constructions." Thèse de troisième cycle, Université Paris VII, 1989.

[14] Ch. Paulin-Mohring. "Extracting F_ω's programs from proofs in the Calculus of Constructions." Proceedings of POPL 1989.

[15] F.Pfenning, P. Lee. "LEAP: A Language with Eval And Polymorphism". ERGO Report 88-065 CMU, Pittsburgh, USA, 1988.

[16] B. Pierce, S.Dietzen, S. Michaylov. "Programming in Higher-Order Typed Lambda-Calculi." Report CMU-CS-89-111. CMU, Pittsburgh, USA, 1989.

[17] P. Weis et al. "The CAML Reference Manual, Version 2.6." Projet FORMEL, INRIA-ENS, March 1989.

Elf: A Language for Logic Definition and Verified Metaprogramming

Preliminary Version

Frank Pfenning

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Internet: `fp@cs.cmu.edu`

Abstract

We describe Elf, a metalanguage for proof manipulation environments that are independent of any particular logical system. Elf is intended for meta-programs such as theorem provers, proof transformers, or type inference programs for programming languages with complex type systems. Elf unifies logic definition (in the style of LF, the Edinburgh Logical Framework) with logic programming (in the style of λProlog). It achieves this unification by giving *types* an operational interpretation, much the same way that Prolog gives certain formulas (Horn-clauses) an operational interpretation. Novel features of Elf include: (1) the Elf search process automatically constructs terms that can represent object-logic proofs, and thus a program need not construct them explicitly, (2) the partial correctness of meta-programs with respect to a given logic can be expressed and proved in Elf itself, and (3) Elf exploits Elliott's unification algorithm for a λ-calculus with dependent types.

1 Introduction

There is a wide variety of deductive systems considered in computer science today (for example Hoare logics, type theories, type deduction systems, operational semantics specifications, first- and higher-order intuitionistic and classical logics). Mechanized support for deduction in a variety of logics and type theories has been the subject of much research (see, for example, Automath [6], LCF [12], HOL [13], Calculus of Constructions [5], Isabelle [25], NuPrl [2]).

In [15], Harper, Honsell, and Plotkin present LF (the Edinburgh Logical Framework) as a general metatheory for the definition of logics. LF provides a uniform way of encoding a logical language, its inference rules and its proofs. In [1], Avron, Honsell, and Mason give a variety of examples for encoding logics in LF. Griffin's EFS (Environment for Formal Systems, see [14]) is an implemen-

[1]This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

[2]This preliminary version appeared in the Proceedings of the Fourth Annual Symposium on Logic in Computer Science, IEEE Computer Society Press, pages 313–322, June 1989.

tation that allows definition of logics and interactive theorem proving in LF. EFS provides a nice syntactic environment, but lacks meta-programming support, this is, it lacks a metalanguage for programming theorem proving, type inference, proof transformation, and similar tasks. We believe that good meta-programming support is essential to obtain adequate theorem proving assistance in an environment that is basically independent of any particular logical system. A signature that defines a logic in LF simply does not contain enough information—it specifies an inference system, but not a useful theorem prover.

In this paper we describe Elf, a metalanguage intended for theorem proving and proof manipulation environments. Its range of applications therefore include the range of applications of such environments (as indicated above). The basic idea behind Elf is to unify logic definition (in the style of LF) with logic programming (in the style of λProlog, see [22, 21, 24]). It achieves this unification by giving *types* an operational interpretation, much the same way that Prolog gives certain formulas (Horn-clauses) an operational interpretation.

Here are some of the salient characteristics of this unified approach to logic definition and metaprogramming. First of all, the Elf search process automatically constructs terms that can represent object-logic proofs, and thus a program need not construct them explicitly. This is in contrast to logic programming languages, where executing a logic program corresponds to theorem proving in a meta-logic, but a meta-proof is never constructed or used and it is solely the programmers responsibility to construct object-logic proofs. Secondly, Elf avoids the undesirable operational behavior of meta-programs that sometimes arises from encoding a logic in higher-order logic as done in Isabelle [25] and by Felty and Miller in [11] (see the example in Section 5.3). Finally, the partial correctness of meta-programs with respect to a given logic can be expressed and proved by Elf itself (see the example in Section 5.2). This creates the possibility of deriving verified meta-programs through theorem proving in Elf (see Constable, Knoblock & Bates [4], Knoblock & Constable [19] or Howe [17] for other approaches).

The base language $\lambda_{\Pi\Sigma}$ for Elf is the LF type theory (a simply typed λ-calculus extended to allow dependent function types), enriched with strong sums (which we prefer to call dependent products). Unlike in λProlog, no restriction on goals or programs needs to be made: the completeness theorem for our abstract interpreter guarantees that goal-directed search is complete. Thus, Elf is a language in the spirit of Miller, Nadathur, Pfenning and Scedrov's [21] definition of an abstract logic programming language, though it is not based on *logic* but on the $\lambda_{\Pi\Sigma}$ *type theory*.

The abstract interpreter, formulated as a transition system, is described in Section 3. Transitions correspond either to unification steps or steps in a goal-directed search for a term of the given type. The unification steps are based on an extensions of Elliott's unification algorithm on terms in the LF type theory (see [8] and [7]). The non-deterministic interpreter is made practical by a commitment to depth-first search, a distinction between *open* and *closed* judgments and *dynamic* and *static* constants, and the addition of the cut search directive familiar from Prolog (see Section 4). The language environment includes a notion of module and a term and type-inference algorithm that makes it much more palatable as an implementation language. Experience with λProlog shows that the commitment to depth-first search results in a useful programming language, even though the underlying unification problem is in general only semi-decidable. In Section 5 we give excerpts from some example Elf programs. An implementation of Elf in Common Lisp is currently in progress in the framework of the Ergo project at Carnegie Mellon University.

2 The base language $\lambda_{\Pi\Sigma}$

The base language for Elf, $\lambda_{\Pi\Sigma}$, is the type theory of LF [15], λ_Π, enriched by a Σ type constructor. The motivation for extending the LF type theory by Σ are discussed in Section 4.4. We are excluding the λ type family constructor as a matter of convenience—in the formulation of LF in [15], λ at the level of types does not appear in normal forms of types and thus seems essential only for the formulation of type inference algorithms. To simplify the presentation we consider α-convertible terms to be identical and also assume that all constants in a signature and variables in a context are distinct.

2.1 Syntax

There are five syntactic categories, just as in λ_Π. In order to be consistent with the notation in [15] we overloaded the symbol Σ to stand for signatures and also be used as a type constructor. It should always be obvious which one is meant. We will use M and N to stand for terms, A and B to stand for types, and K to stand for kinds.

$$
\begin{array}{ll}
\text{Signatures} & \Sigma ::= \langle\rangle \mid \Sigma, c{:}K \mid \Sigma, c{:}A \\
\text{Contexts} & \Gamma ::= \langle\rangle \mid \Gamma, x{:}A \\
\text{Kinds} & K ::= \text{Type} \mid \Pi x{:}A\,.\,K \\
\text{Type Families} & A ::= c \mid A\,M \mid \Pi x{:}A\,.\,B \mid \Sigma x{:}A\,.\,B \\
\text{Terms} & M ::= c \mid x \mid \lambda x{:}A\,.\,M \mid M\,N \\
 & \qquad\quad \mid (M, N) \mid \text{fst}\,M \mid \text{snd}\,M
\end{array}
$$

We will use the abbreviations $A \to B$ for $\Pi x{:}A\,.\,B$ when x is not free in B, and $A \times B$ for $\Sigma x{:}A\,.\,B$ when x is not free in B. $[M/x]N$, $[M/x]A$, and $[M/x]K$ are our notation for substitution, renaming bound variables if necessary to avoid name clashes. Atomic types C are types that begin with neither a Π nor a Σ.

2.2 Typing rules

We adopt the presentation from [15] and add the following rules for dependent products (strong sums):

$$\frac{\Gamma \vdash_\Sigma A \in \text{Type} \qquad \Gamma, x{:}A \vdash_\Sigma B \in \text{Type}}{\Gamma \vdash_\Sigma \Sigma x{:}A\,.\,B \in \text{Type}}$$

$$\frac{\Gamma \vdash_\Sigma M \in A \qquad \Gamma \vdash_\Sigma N \in [M/x]B}{\Gamma \vdash_\Sigma (M, N) \in \Sigma x{:}A\,.\,B}$$

$$\frac{\Gamma \vdash_\Sigma M \in \Sigma x{:}A\,.\,B}{\Gamma \vdash_\Sigma \text{fst}\,M \in A} \qquad \frac{\Gamma \vdash_\Sigma M \in \Sigma x{:}A\,.\,B}{\Gamma \vdash_\Sigma \text{snd}\,M \in [\text{fst}\,M/x]B}$$

2.3 Conversion rules

When choosing a notion of conversion for $\lambda^{\Pi\Sigma}$ there is an unfortunate tradeoff. From a practical point of view (both for the encoding of inference systems in $\lambda^{\Pi\Sigma}$ and for the implementation of Elf) one would like $\beta\eta$-reduction and the rules for surjective pairing. Unfortunately, the Church-Rosser property for well-typed terms under this notion of reduction is still open, though we conjecture that it holds. One can weaken the notion of conversion and obtain the Church-Rosser property by omitting η and π, but a complete abstract interpreter for Elf using this weaker notion would have to be unduly complex. For λ^{Π} (without pairing) these tradeoffs already exist and are discussed at some length in [15]. Here are the reduction rules in question:

$$\begin{array}{lll}
\beta & (\lambda x{:}A\,.\,M)\,N \longrightarrow_{\beta} [N/x]M & \\
\pi^1 & \mathrm{fst}\,(M,N) \longrightarrow_{\pi^1} M & \\
\pi^2 & \mathrm{snd}\,(M,N) \longrightarrow_{\pi^2} N & \\
\eta & (\lambda x{:}A\,.\,M\,x) \longrightarrow_{\eta} M & \text{if } x \text{ not free in } M \\
\pi & (\mathrm{fst}\,M, \mathrm{snd}\,M) \longrightarrow_{\pi} M &
\end{array}$$

A term is *normal form* if none of the reduction rules above apply, and a term is *strongly normalizing* if every sequence of reductions terminates.

We use $\approx$ for *strong conversion*, that is, all reductions may be used an arbitrary number of times in either direction at any location in a term, type, or kind. *Weak conversion* is generated the same way, but only from β, π^1, and π^2. In either case, a reinterpretation of the type and kind conversion rules in the presentation of λ^{Π} is required to encompass a larger set of conversions.

2.4 Some properties of $\lambda^{\Pi\Sigma}$

$\lambda^{\Pi\Sigma}$ inherits some, but not all of its properties from LF.

Theorem 1 (Basic properties of $\lambda^{\Pi\Sigma}$ under weak conversion.)

1. *All terms are Church-Rosser.*
2. *All well-typed terms are strongly normalizing.*
3. *Type-checking and kind-checking is decidable.*

Proof: The proof is an extension of the one in [15]. Church-Rosser for this weak notion of conversion can be proved as in [15], since it holds for all, and not only for well-typed terms. To prove strong normalization, we translate both types and terms from $\lambda^{\Pi\Sigma}$ into terms in a simply typed λ-calculus with products and explicit types. It follows from a the strong normalization theorem by Troelstra [28] for a simply typed λ-calculus (even including η and surjective pairing) that such terms are strongly normalizing which in turn implies this for the original terms with dependent types.

$\square$

Under strong conversion, only the proof of strong normalization for well-typed terms can be extended in a straightforward way, as indicated in the proof sketch above. The Church-Rosser property now fails in general, but we conjecture that it holds for well-typed terms. For the remainder of this paper we will use the notion of strong conversion, since it is desirable in practice and the basis for our implementation. The theorems are therefore qualified by an assumption about the Church-Rosser property for $\lambda_{\Pi\Sigma}$. The abstract interpreters and theorems could be modified to suit the notion of weak reduction, but the additional complexity introduced seems unwarranted.

One of the properties that does *not* hold for $\lambda_{\Pi\Sigma}$ due to the presence of products is uniqueness of types, that is, a given well-typed term M may have many different types. This may seem like a basic flaw, but it does not lead to problems in Elf where computation originates from the structure of types rather than the structure of terms.

3 An abstract interpreter

Before turning to the formal definition of the abstract interpreter, let us outline why Elf is more than just a theorem prover, but a programming language, similar in many respects to logic programming languages.

The basic idea behind turning a logic into a logic programming language is to identify two sets of formulas: legal goals and legal programs. Many factors may influence the choice of these sets, but we would like to single out particularly important criterion (as argued in [21]): a (non-deterministic) abstract interpreter should be able to perform goal-directed search in such a way that every legal goal that is a theorem in the underlying logic, will succeed. It is interesting to note that this condition is independent of any notion of unification and thus encompasses *constraint logic programming* (see Jaffar & Lassez [18]).

Here we are in a similar situation, turning a type theory into a programming language. The basic idea is to give *types* an operational interpretation much in the same way that formulas are given an operational interpretation in a logic programming language.

Informally, this operational interpretation is as follows. A goal $z \in \Sigma x{:}A \,.\, B$ should succeed, if the goals $x \in A$ and $y \in B$ both succeed, and z is the pair (x, y). Note that x may occur in B, and that therefore the two subgoals may not be independent. Thus x may serve as a "logical variable" except that it may also range over *proofs* constructed by the interpreter, something not possible in logic programming languages.

A goal $z \in \Pi x{:}A \,.\, B$ should succeed, if the goal $y \in B$ succeeds under the assumption that x has type A, and z is the abstraction $\lambda x{:}A \,.\, y$ (where x may occur free in y, since x may occur free in B). This does not correspond to any construct in a Horn-clause logic, but the Π type construction serves the role of the $\forall$ and $\supset$ connectives in a hereditary Harrop logic (see [24]).

The natural criterion for the choice of legal goal and program types is slightly stronger here, since we would also like to ensure that *all* terms of the given type can be found using goal-directed search, interpreting Σ and Π as outlined above. Actually we can only require that for any term M of the given type A, we can find a term N such that $N \approx M$. Surprisingly, this criterion is satisfied if we admit *all* types of $\lambda_{\Pi\Sigma}$ as goals and programs. We will not formalize and prove this fact as a separate theorem, since it follows rather directly from the completeness of the abstract interpreter. Intuitively, this is due to the presence of strong sums, while a (logical) existential quantifier is only a weak sum.

$$\frac{\Gamma \vdash_\Sigma M \in A}{\Gamma \Vdash_\Sigma M \in A} \qquad \frac{\Gamma \vdash_\Sigma M \in A \qquad M \approx N \qquad \Gamma \vdash_\Sigma N \in A}{\Gamma \Vdash_\Sigma M \doteq N \in A}$$

$$\Gamma \Vdash_\Sigma M \in C \supset M \in C \qquad \frac{\Gamma \Vdash_\Sigma N\, N_0 \in [N_0/x]B \supset M \in C \qquad \Gamma \Vdash_\Sigma N_0 \in A}{\Gamma \Vdash_\Sigma N \in \Pi x{:}A \,.\, B \supset M \in C}$$

$$\frac{\Gamma \Vdash_\Sigma \mathrm{fst}\, N \in A \supset M \in C}{\Gamma \Vdash_\Sigma N \in \Sigma x{:}A \,.\, B \supset M \in C} \qquad \frac{\Gamma \Vdash_\Sigma \mathrm{snd}\, N \in [\mathrm{fst}\, N/x]B \supset M \in C}{\Gamma \Vdash_\Sigma N \in \Sigma x{:}A \,.\, B \supset M \in C}$$

$$\frac{\Gamma, x{:}A \Vdash_\Sigma F}{\Gamma \Vdash_\Sigma \forall x{:}A \,.\, F} \qquad \frac{\Gamma \Vdash_\Sigma [M/x]F \qquad \Gamma \Vdash_\Sigma M \in A}{\Gamma \Vdash_\Sigma \exists x{:}A \,.\, F}$$

Figure 1: Deduction rules for the state logic

3.1 A state logic

The inference system for type deduction in $\lambda_{\Pi\Sigma}$ defines a number of judgments, such as convertibility, or $\Gamma \vdash_\Sigma M \in A$. However, the structure of conclusions on the right-hand of $\vdash_\Sigma$ is not expressive enough to describe the states of an abstract interpreter for Elf. In order to gain this expressive power, we generalize $\lambda_{\Pi\Sigma}$ by introducing a new judgment $\Gamma \Vdash_\Sigma F$ with a much richer language for conclusions F. We refer to the conclusions F as *formulas*. We use the letter C to stand for atomic types, which in $\lambda_{\Pi\Sigma}$ have the form $c\, M_1 \ldots M_n$.

$$\begin{array}{rl} F ::= & M \doteq N \in A \mid M \in A \mid N \in A \supset M \in C \\ & \mid T \mid F_1 \wedge F_2 \mid \forall x{:}A \,.\, F \mid \exists x{:}A \,.\, F \end{array}$$

The first line contains the formulas that are considered atomic. Except for atomic $\doteq$ and $\supset$ formulas, this is very close to the unification logic introduced in [26], and it is used in a very similar fashion. The restricted form of implication is used to describe the backchaining in the abstract interpreter. The inference system in Figure 1 defines the judgment $\Vdash_\Sigma$. Note that these inference rules do not define a search process or strategy, merely a judgment—it is the abstract interpreter in Section 3.2 which defines a complete (non-deterministic) search procedure.

The basic property of the state logic is summarized in the following theorem. Of course, the completeness with respect to the atomic formulas that are also judgments in $\lambda_{\Pi\Sigma}$ is obvious.

Theorem 2 (Soundness of state logic)

1. *If* $\Gamma \Vdash_\Sigma M \in A$ *then* $\Gamma \vdash_\Sigma M \in A$.

2. *If* $\Gamma \Vdash_\Sigma M \doteq N \in A$ *then* $\Gamma \vdash_\Sigma M \in A$, $M \approx N$, *and* $\Gamma \vdash_\Sigma N \in A$.

The proof is by straightforward inductions on the form of deductions of $\Gamma \Vdash_\Sigma F$.

$$
\begin{array}{rll}
G_\Sigma: & M \in \Sigma x{:}A\,.\,B \Longrightarrow \exists x{:}A\ \exists y{:}B\,.\,M \doteq (x,y) \in \Sigma x{:}A\,.\,B \wedge x \in A \wedge y \in B \\
G_\Pi: & M \in \Pi x{:}A\,.\,B \Longrightarrow \forall x{:}A\ \exists y{:}B\,.\,M\,x \doteq y \in B \wedge y \in B \\
G^1_{\mathrm{Atom}}: & M \in C \qquad\quad \Longrightarrow x \in A \supset M \in C \quad \text{where } M \in C \text{ is in the scope of } \forall x{:}A. \\
G^2_{\mathrm{Atom}}: & M \in C \qquad\quad \Longrightarrow c_0 \in A \supset M \in C \quad \text{where } c_0{:}A \text{ in } \Sigma.
\end{array}
$$

$$
\begin{array}{rll}
D_\Pi: & N \in \Pi x{:}A\,.\,B \supset M \in C & \Longrightarrow \exists x{:}A\,.\,(N\,x \in B \supset M \in C) \wedge x \in A \\
D^1_\Sigma: & N \in \Sigma x{:}A\,.\,B \supset M \in C & \Longrightarrow \mathrm{fst}\,N \in A \supset M \in C \\
D^2_\Sigma: & N \in \Sigma x{:}A\,.\,B \supset M \in C & \Longrightarrow \mathrm{snd}\,N \in [\mathrm{fst}\,N/x]B \supset M \in C \\
D_{\mathrm{Atom}}: & N \in c\,N_1 \ldots N_n \supset M \in c\,M_1 \ldots M_n & \\
& \qquad \Longrightarrow N_1 \doteq M_1 \in A_1 \wedge \ldots \wedge N_n \doteq M_n \in A_n \wedge N \doteq M \in A &
\end{array}
$$

Figure 2: Transition of non-deterministic abstract interpreter

3.2 A first non-deterministic abstract interpreter

We now present the non-deterministic transition system on formulas in the state logic that defines our first abstract interpreter. The rewrites may be applied at any occurrence in the state formula. Given implicitly is a signature Σ. The transition system is organized into classes of transitions, each class dealing with different atomic formulas. Since some information is needed by different components of the abstract interpreter, the state formula F will contain some seemingly redundant information.

3.2.0.1 Goal transitions G. These four transitions (see G_Σ, G_Π, G^1_{Atom}, and G^2_{Atom} in Figure 2) analyze formulas of the form $M \in A$. Note that x may appear free in the type B, and remember that C stands for an atomic type.

3.2.0.2 Backchaining transitions D. The final two transitions in the previous group create implications which are now further analyzed by the transitions D_Π, D^1_Σ, D^2_Σ, and D_{Atom} in Figure 2). In a Horn-clause logic they can be formulated more easily, since the necessary subgoals are immediately available in the body of a clause—here subgoals have to be constructed. Of course, the actual implementation can be more efficient.

In D_{Atom}, the types $A_1, \ldots, A_n$ and A are determined from the kind of c in the signature. Note that A_i may contain N_j for $j < i$, and A may contain all terms N_i.

3.2.0.3 Unification transitions U. Unfortunately, space does not permit to include a presentation of the unification transitions, which are discussed in [8]. There are two extensions to Elliott's algorithm required here, both of which have been described for the simply typed λ-calculus and carry over to $\lambda_{\Pi\Sigma}$ in a straightforward way: (1) the dependency of universal and existential quantifiers must be taken into account without Skolemization (see Miller [23]), and (2) the algorithm must deal with products (see Elliott [7]).

We write $\Longrightarrow^*$ for the reflexive and transitive closure of the transition relation $\Longrightarrow$. At this point we are ready to formulate a first preliminary soundness and completeness theorems for the abstract interpreter with respect to the state logic. We will later refine $\Longrightarrow^*$, since the interpreter as stated so far is still too non-deterministic. We are omitting here soundness and completeness theorems for this first abstract interpreter, since they are subsumed by Theorem 3.

3.3 Open and closed judgments

We now introduce the important concepts of *open* and *closed* judgments. Judgments are represented in LF (and Elf) as type families, so they are corresponding notions of open and closed types. This step towards a practical programming language is still fully justified by the underlying type theory and does not introduce any incompleteness. Intuitively, we are willing to tolerate free variables of open type in proofs, but no free variables of closed type. In an encoding of first-order logic as in [15], $\phi\,$true would be a closed judgment, while i (representing the domain of individuals) would be an open judgment. It is the programmer's responsibility to annotate constants in the signature as `open` or `closed`, but a convenient defaulting mechanism is provided.

Definition 1 *A state F is* solved *iff*

1. *there are no implicational atomic subformulas in F,*
2. *every atomic subformula $M \doteq N \in A$ is in* solved form[1], *and*
3. *for every atomic subformula $M \in A$, M is an existentially quantified variable and A is open.*

We now restrict our abstract interpreter to account for open and closed judgments by placing some of the burden for completeness of the unification transitions. Let $\Longrightarrow_U$ be the restriction of $\Longrightarrow$ by restricting uses of transitions rules G^1_{Atom} and G^2_{Atom} to the case where c is a closed type family. Let $\Longrightarrow^*_U$ be the reflexive and transitive closure of $\Longrightarrow_U$.

Theorem 3 *Given a signature Σ with open type constants $\mathcal{O}$ and a type A with free variables $y_1{:}A_1, \ldots, y_n{:}A_n$. Under the assumption of the weak Church-Rosser property for $\lambda^{\Pi\Sigma}$ under strong conversion, $\Vdash_\Sigma \exists y_1{:}A_1 \ldots \exists y_n{:}A_n\ \exists x{:}A\ .\ y_1 \in A_1 \wedge \ldots \wedge y_n \in A_n \wedge x \in A \quad \Longrightarrow^*_U \quad \Vdash_\Sigma F$ for some solved F iff there are $N_1, \ldots, N_n$ and M such that $\vdash_\Sigma M \in [N_1/y_1] \ldots [N_n/y_n]A$ and any free variable in $N_1, \ldots, N_n$ and M has open type.*[2]

The proof is constructive, that is, gives explicit transformations of transition sequences to deductions in the state logic and vice versa. It also requires the completeness of higher-order unification, since open types are not analyzed as goals with respect to the signature.

As discussed earlier, a different version of this theorem for a modified interpreter can be given for $\lambda^{\Pi\Sigma}$ under weak conversion, but this modified theorem is not practically motivated. Note that only completeness depends on the Church-Rosser property, not soundness.

[1]Pairs in solved form are guaranteed to have solutions. Elliott's unification algorithm uses the criterion that both M and N are "flexible" (see [8]).

[2]The abstract interpreter satisfies a stronger condition: it characterizes *all* terms $M \in A$. Stating a theorem to this effect would require a discussion of higher-order preunification, which is beyond the scope of this paper.

4 The Elf language and interpreter

We now proceed to turn the abstract interpreter into a practical interpreter, following the ideas underlying λProlog. These commitments and extensions are motivated by the experience with Prolog and λProlog, and completeness is lost. This step leads to Elf as a true *programming language* in which one can write theorem proving programs, rather than a logic-independent theorem prover (which we believe to be a problem too difficult for a general, complete solution).

4.1 Depth-first search

Search through the program is committed to be depth-first. This means that the interpreter goes through the state formula from left to right until it encounters an atomic formula F.

1. If F matches the left-hand side of G_Σ or G_Π, that rule is applied.

2. If F is of the form $M \in C$ for atomic and closed C, it applies G^1_{Atom} to the innermost quantifier $\forall x{:}A$ such that A is closed and $M \in C$ is in its scope. On backtracking, the next further universal quantifier is considered, etc., until all have been considered. Finally the *current signature* Σ (see Section 4.2) is scanned from left to right, applying G^2_{Atom} to declarations $c_0 \in A$ for closed A.

3. If F is an implication, we apply rule D_Π if it matches. If D^1_Σ applies, we use it, and use D^2_Σ on backtracking. Finally we apply D_{Atom} if both atomic types begin with the same constant. Otherwise we backtrack over previous choices.

4. If F is T, an equality in solved form, or $M \in C$ for open C, we pass over it, looking for the next atomic formula. Thus, equalities in solved form are constraints in the sense of Jaffar & Lassez [18].

5. If F is an equality not in solved form, we call the unification algorithm on the whole state. Unification may fail (upon which we backtrack), not terminate, or replace the given equality by a conjunction of solved equalities, with T representing the empty conjunction. On backtracking, the unifier will enumerate more solutions, which is necessary since unique most general unifiers do not exist for $\lambda_{\Pi\Sigma}$ in general.

In the remainder of the paper, we will refer to the program defined by cases 1 through 4 as the *goal interpreter*, case 5 defines the *unifier*.

4.2 Dynamic and static constants

In the interpreter as given above we have used the notion of *current signature*. Signatures are the basic unit of programs, and they serve two purposes. They are necessary for unification (which includes term- and type-checking and inference, see Section 4.3) and for the goal-directed search performed by the goal interpreter. If all constants were visible to the goal interpreter, this would lead to very undesirable behavior. For example $\supset\!\mathrm{E} : \Pi A{:}o\ .\ \Pi B{:}o\ .\ \vdash A \supset B \to \vdash A \to \vdash B$

would apply to any goal of the the form $\vdash C$ and lead to very undirected search. However, the type of the constant $\supset$E must be available to the unifier. Therefore term constants (such as $\supset$E) may be declared as *dynamic* or *static*. A dynamic constant will be used by the interpreter when visible according to the module visibility rules [3]. A static constant will never be used by the goal interpreter. The type of both dynamic and static constants will be visible to the unifier.

4.3 Term and type inference

Using the LF type theory or $\lambda_{\Pi\Sigma}$ without term and type inference can be extremely cumbersome, since much information would have to be given that could be inferred. The basic mechanism for term inference in λ_{Π} is described in [8]. It is extremely important to note that term inference, as defined by Elliott, does *not* require general theorem proving, since it leaves free variables of closed type uninstantiated, even though there may not be any terms without free variables of such a type. Instead, it relies entirely on unification on terms in $\lambda_{\Pi\Sigma}$. Unfortunately, unification on $\lambda_{\Pi\Sigma}$ (and λ_{Π}) is only semi-decidable, and the term-inference problem is only semi-decidable as well. Therefore, a resource bound is put on term inference, and it may return three answers (yes, no, or maybe). In case of a "maybe" the user can add more type information to his program. Experience with higher-order unification suggests that $\lambda_{\Pi\Sigma}$-unification should be able to handle most practical examples of term inference rather easily, so a small resource bound should suffice. The additional complexity of types over terms is small and term-inference can be extended easily to type inference. A more serious practical problem is that of ambiguity: omitted types and terms can often be restored in a number of incompatible ways. Currently, we require more information from the user in such a case.

4.4 Σ-types and queries

Let us return to the motivation for including dependent products in the language. Firstly, Σ-types and the ability to form pairs of terms are a matter of convenience, in the same way conjunction in logic programming is a matter of convenience, though not strictly necessary (one could use nested implications). Secondly, products combined with polymorphism significantly strengthen Elf as a representation and metaprogramming language to handle the common case of a object languages with binding constructs of variable arity (see [27]). For example, a natural encoding of Hoare logic in LF as given in Section 4.10 of [1] must be restricted to a fixed number of registers—a restriction that can be dropped in $\lambda_{\Pi\Sigma}$ with polymorphism. Thirdly, Σ-types can introduce "constants" that are local to a module as Σ-quantified variables, something not possible in λProlog. Finally, products are important in queries, where they can be used to hide information (thinking of Σ as an existential quantifier) and to express several goals to be satisfied simultaneously.

5 Examples

We give excerpts from some programs that highlight some of the unique features of Elf that set it apart from related languages such as λProlog. Some of the types given in these examples could be

[3] similar to the ones in λProlog, see [20]. Another related approach to structuring of theories may be found in [16].

inferred. A feature used in the examples that has not yet been discussed is a very weak form of equality in signatures, $c = M$, which is used exclusively for term and type inference.

5.1 A module defining a first-order logic

We give a condensed signature for a first-order logic in the style of Elf below. It is very close to the LF encoding in [15], with the exception of some annotations. We are now using A and B to stand for formulas, and $\vdash A$ for the judgment that A is true.

$$
\begin{array}{l}
\texttt{static Module}\, \mathit{fol} \\
\quad \begin{array}{rll}
\texttt{open}\; i & : \mathrm{Type} & \text{\% type of terms.} \\
\texttt{open}\; o & : \mathrm{Type} & \text{\% type of propositions.} \\
\texttt{closed}\; \vdash & : o \to \mathrm{Type} & \text{\% type of proofs.}
\end{array} \\[1ex]
\quad \begin{array}{rl}
\bot & : o \\
\neg & : o \to o \\
\wedge, \vee, \supset & : o \to o \to o \\
\forall, \exists & : (i \to o) \to o \\[1ex]
\bot_I{}' & : \Pi C \,.\, \vdash\bot \to \vdash C \\
\neg\mathrm{I}' & : \Pi A \,.\, (\vdash A \to \vdash\bot) \to \vdash\neg A \\
\neg\mathrm{E}' & : \Pi A \,.\, \vdash\neg A \to \vdash A \to \vdash\bot \\
\wedge\mathrm{I}'' & : \Pi A \,.\, \Pi B \,.\, \vdash A \to \vdash B \to \vdash A \wedge B \\
\wedge\mathrm{E}_l'' & : \Pi A \,.\, \Pi B \,.\, \vdash A \wedge B \to \vdash A \\
\wedge\mathrm{E}_r'' & : \Pi A \,.\, \Pi B \,.\, \vdash A \wedge B \to \vdash B \\
\vee\mathrm{I}_l' & : \Pi A \,.\, \Pi B \,.\, \vdash A \to \vdash A \vee B \\
\vee\mathrm{I}_r' & : \Pi A \,.\, \Pi B \,.\, \vdash A \to \vdash B \vee A \\
\vee\mathrm{E}''' & : \Pi C \,.\, \Pi A \,.\, \Pi B \,.\, \vdash A \vee B \to \\
 & \qquad (\vdash A \to \vdash C) \to (\vdash A \to \vdash C) \to \vdash C \\
\supset\mathrm{I}'' & : \Pi A \,.\, \Pi B \,.\, (\vdash A \to \vdash B) \to \vdash A \supset B \\
\supset\mathrm{E}'' & : \Pi A \,.\, \Pi B \,.\, \vdash A \supset B \to \vdash A \to \vdash B \\
\forall\mathrm{I}' & : \Pi A{:}i \to o \,.\, (\Pi x{:}i \,.\, \vdash A\, x) \to \vdash\forall\, A \\
\forall\mathrm{E}' & : \Pi A{:}i \to o \,.\, \vdash\forall A \to (\Pi x{:}i \,.\, \vdash A\, x) \\
\exists\mathrm{I}' & : \Pi A{:}i \to o \,.\, \Pi x{:}i \,.\, \vdash A\, x \to \vdash\exists A \\
\exists\mathrm{E}'' & : \Pi C \,.\, \Pi A{:}i \to o \,.\, (\Pi x{:}i \,.\, \vdash A\, x \to \vdash C) \\
 & \qquad \to \vdash\exists A \to \vdash C
\end{array}
\end{array}
$$

5.2 Miniscoping

Minimizing the scope of quantifiers is an common preprocessing step in theorem provers. This example illustrates how a "clause" (a declaration `dynamic` $c : A$) in Elf can be proved correct by giving a closed term of type A. In practice, one would obtain the proof of $\forall$push$\wedge$ by theorem proving in a programming environment (for example, in the style of Isabelle [25]), an issue beyond the scope of this paper. The excerpt contains two clauses, one that pushes a universal quantifier over a conjunction, and one that pushes a universal quantifier over a disjunction, if the bound variable does not appear free in the right disjunct.

$$
\begin{array}{l}
\forall\mathrm{push}\wedge'' : \Pi A\, \Pi B\, .\, \vdash \forall\, (\lambda x\, .\, A\, x \wedge B\, x) \leftarrow \vdash \forall A \wedge \forall B \\
\forall\mathrm{push}\wedge'' {=} \lambda A\, .\, \lambda B\, .\, \lambda z{:}\vdash \forall A \wedge \forall B\, .\, \forall\mathrm{I}'\, \_\, (\lambda x{:}i\, . \\
\quad \wedge\mathrm{I}''\, (A\, x)\, (B\, x)\, (\forall\mathrm{E}'\, \_\, (\wedge\mathrm{E}'_l\, \_\, z)\, x)\, (\forall\mathrm{E}'\, \_\, (\wedge\mathrm{E}'_r\, \_\, z)\, x))
\end{array}
$$

$$
\begin{array}{l}
\forall\mathrm{push}\vee_l : \vdash \forall\, (\lambda x\, .\, A\, x \vee B) \leftarrow \vdash \forall A \vee B \\
\forall\mathrm{push}\vee_l {=} \lambda z{:}\vdash \forall A \vee B\, .\, \forall\mathrm{I}\, (\lambda x{:}i\, .\, \vee\mathrm{E}\, z \\
\qquad\qquad (\vee\mathrm{I}_l\, \_\, (\lambda y_1\, .\, \forall\mathrm{E}\, y_1\, x))\, (\vee\mathrm{I}_r\, \_\, (\forall A)))
\end{array}
$$

5.3 Proof reduction and normalization

This next example is the implementation of proof reductions for proofs in a first-order intuitionistic logic. Felty showed in [9] how to implement the reductions in λProlog, but at a very heavy price. Each of her clauses contains a proof-checking subgoal that may be very expensive to execute. If her reductions are combined into a normalization procedure, each reduction step must do proof-checking, something that can be avoided in Elf. This is because Elliott's unification algorithm eliminates redundant type-checking (which is proof-checking in an encoded logic) in many cases. The following reductions rules are verified since the type of reduce$'$ guarantees that its second and third argument are proofs of the same theorem. The constants in this example are anonymous and their names will be generated by Elf.

$$
\begin{array}{l}
\mathrm{reduce}'{:}\Pi A\, .\, \vdash A \to \vdash A \to \mathrm{Type} \\
\qquad \mathrm{reduce}\ (\neg\mathrm{E}\, (\neg\mathrm{I}\, M)\, N)\ (M\, N) \\
\qquad \mathrm{reduce}\ (\wedge\mathrm{E}_l\, (\wedge\mathrm{I}\, M\, N))\ M \\
\qquad \mathrm{reduce}\ (\wedge\mathrm{E}_r\, (\wedge\mathrm{I}\, M\, N))\ N \\
\qquad \mathrm{reduce}\ (\vee\mathrm{E}\, (\vee\mathrm{I}_l\, \_\, M)\, N_1\, N_2)\ (N_1\, M) \\
\qquad \mathrm{reduce}\ (\vee\mathrm{E}\, (\vee\mathrm{I}_r\, \_\, M)\, N_1\, N_2)\ (N_2\, M) \\
\qquad \mathrm{reduce}\ (\supset\mathrm{E}\, (\supset\mathrm{I}\, M) N)\ (M\, N) \\
\qquad \mathrm{reduce}\ (\forall\mathrm{E}\, (\forall\mathrm{I}\, M)\, T)\ (M\, T) \\
\qquad \mathrm{reduce}\ (\exists\mathrm{E}\, N\, (\exists\mathrm{I}\, T\, M))\ (N\, T\, M)
\end{array}
$$

The commutative reductions can be formulated very generally (and non-deterministically). The declaration as `static` ensures that this is not used by the goal interpreter, only by the type-checker.

$$
\begin{array}{l}
\texttt{static}\ \mathrm{cred}''' : \Pi A\, \Pi B\, \Pi C\, \Pi F{:}(\vdash C \to \vdash C)\, . \\
\quad \Pi M{:}\vdash A \vee B\, .\, \Pi N_1\, .\, \Pi N_2\, . \\
\qquad \mathrm{reduce}'\ C\ (F\, (\vee\mathrm{E}\, M\, N_1\, N_2)) \\
\qquad\quad (\vee\mathrm{E}\, M\, (\lambda x\, .\, F\, (N_1\, x))\, (\lambda y\, .\, F\, (N_2\, y)))
\end{array}
$$

We can then use specializations of this general rule dynamically in the normalization program. The two specializations listed below apply in the case of a $\supset$E directly preceded by an $\vee$E, either in the left or right premise. The types of cr_1 and cr_2 are inferred by the type inference algorithm. One can write similar specializations for $\vee$E followed by other elimination rules that form a maximal segment. Note the conciseness of this formulation over the many rules one must state in λProlog, and all of which are instances of the general transformation cred.

$$\begin{array}{ll}\texttt{dynamic} & \mathrm{cr}_1{=}\mathrm{cred}\,(\lambda x\,.\,(\supset\!\mathrm{E}\,x\,\_)) \\ \texttt{dynamic} & \mathrm{cr}_2{=}\mathrm{cred}\,(\lambda x\,.\,(\supset\!\mathrm{E}\,\_\,x))\end{array}$$

5.4 Other examples

Another important application is to mechanize type-checking for programming languages with complex type systems. The property of a program to be well-typed in such a language can be formalized in Elf as an inference system in the style of LF. This does not immediately lead to a type-checking algorithm, unless there is also a theorem prover for some of the more complex relations (like subtypes). Often the search space for such relations is linear and a decision procedure can be given immediately, that is, the signature itself may be used dynamically.

Natural deduction theorem provers such as Gentzen (see Beeson [2]) or those proposed by Felty and Miller [11] can also be expressed very naturally in Elf. The extraction of programs from proofs is another example of the kind of algorithm that can easily be implemented in Elf. In many of these examples, the implementations are related to λProlog implementations of a similar flavor. The main added advantages of Elf are (1) the program does not need to keep track of *proofs* explicitly—that is done by the Elf interpreter itself (without performance penalty, when proofs are not used), (2) the partial correctness of many programs can be guaranteed by Elf, and (3) the operational behavior of Elf is much better when explicit type-checking or proof-checking would be required in λProlog (as is frequently necessary in the programs obtained by translating LF signatures into λProlog programs as outlined by Felty in [11]).

6 Implementation and further work

An implementation of Elf in Common Lisp is in progress in the framework of the Ergo project at Carnegie Mellon University. Among the important optimizations not mentioned above are (1) signatures are transformed and stored in a hash-table indexed by the type families they define which allows fast backchaining in the style of Prolog, and (2) term construction can often be avoided if the constructed term would not be used (which is frequently the case when a theorem prover is called, since proofs are often irrelevant). The implementation also contains a few extra-logical primitives such as cut, `read` and `write`, and a module system, similar to the one in λProlog.

Extensions we are considering concern polymorphism (which is included in the implementation, but treated in an incomplete way), a stronger notion of definitional equality including δ-reductions, and the embedding of Elf in a general proof development and transformation environment. We are also considering a sublanguage of Elf along the lines of Felty and Miller's L_λ [10] for which unification would be decidable.

Acknowledgments

I would like to thank Ken Cline, Conal Elliott, Amy Felty, and Dale Miller for helpful discussions concerning the subject of this paper.

References

[1] Arnon Avron, Furio A. Honsell, and Ian A. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, Scotland, June 1987.

[2] M. Beeson. Some applications of Gentzen's proof theory in automated deduction. Submitted, 1988.

[3] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[4] Robert L. Constable, Todd Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1(3):285–326, 1984.

[5] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[6] N. G. de Bruijn. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press, 1980.

[7] Conal Elliott. Some extensions and applications of higher-order unification: A thesis proposal. Ergo Report 88–061, Carnegie Mellon University, Pittsburgh, June 1988. Thesis to appear June 1989.

[8] Conal Elliott. Higher-order unification with dependent types. In *Rewriting Techniques and Applications*, pages 121–136. Springer-Verlag LNCS 355, April 1989.

[9] Amy Felty. Implementing theorem provers in logic programming. Technical Report MS-CIS-87-109, University of Pennsylvania, Philadelphia, December 1987.

[10] Amy Felty and Dale Miller. A metalanguage for type checking and inference. Manuscript, November 1988.

[11] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 61–80, Berlin, May 1988. Springer-Verlag LNCS 310.

[12] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF.* Springer-Verlag LNCS 78, 1979.

[13] Mike Gordon. Hol: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, July 1985.

[14] Timothy G. Griffin. An environment for formal systems. Technical Report 87-846, Department of Computer Science, Cornell University, Ithaca, New York, June 1987.

[15] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. Submitted. A preliminary version appeared in *Symposium on Logic in Computer Science*, pages 194–204, June 1987, January 1989.

[16] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Structure and representation in LF. In *Fourth Annual Symposium on Logic in Computer Science*, pages 226–237. IEEE, June 1989.

[17] Douglas J. Howe. Computational metatheory in Nuprl. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction, Argonne, Illinois*, pages 238–257, Berlin, May 1988. Springer-Verlag LNCS 310.

[18] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 111–119. ACM, January 1987.

[19] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *First Annual Symposium on Logic in Computer Science, Cambridge, Massachusetts*, pages 237–248. IEEE Computer Society Press, June 1986.

[20] Dale Miller. A logical analysis of modules for logic programming. *Journal of Logic Programming*, 1988. To appear.

[21] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Journal of Pure and Applied Logic*, 1988. To appear. Available as Ergo Report 88–055, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[22] Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Second Annual Symposium on Logic in Computer Science*, pages 98–105. IEEE, June 1987.

[23] Dale A. Miller. Unification under mixed prefixes. Unpublished manuscript, 1987.

[24] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, pages 810–827, Cambridge, Massachusetts, August 1988. MIT Press.

[25] Lawrence C. Paulson. The representation of logics in higher-order logic. Technical Report 113, University of Cambridge, Cambridge, England, August 1987.

[26] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming, Snowbird, Utah*, pages 153–163. ACM Press, July 1988. Also available as Ergo Report 88–048, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[27] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[28] Anne S. Troelstra. Strong normalization for typed terms with surjective pairing. *Notre Dame Journal of Formal Logic*, 27(4):547–550, October 1986.

Implementing General Recursion in Type Theory

Colin Phillips

Department of Artificial Intelligence, Edinburgh University

Abstract

This paper describes an implementation of Nordström's acc type in Oyster, a version of the NUPRL system. Amongst the problems involved in using Martin-Löf's type theory as a program language are those that arise because of computational redundancy and the lack of an adequate mechanism for dealing with general recursion. Bengt Nordström has proposed an extension to type theory - the Acc type - which provides a possible mechanism for dealing with these problems. This paper describes the mechanism for implementing the Acc type in Oyster. There is also a discussion of a number of examples including a derivation of the quicksort program and the lambo program.

1 Introduction

This paper describes an implementation of Nordström's Acc type [4] in Oyster [3], a version of the NUPRL system [2].

It is widely recognised that there are a number of problems involved in using Martin-Löf's type theory as a program language. One problem arises because of computational redundancy. If one attempts to extract a program from a proof in type theory the resulting extract term may contain, in addition to the program one wants, additional information which pertains to the proof that the program satisfies the specification it is intended to satisfy. Secondly, although Martin-Löf's type theory contains a mechanism for dealing with primitive recursion over natural numbers and other inductive sets , there is no procedure for dealing with general recursion. Thirdly, a problem that I will not in fact consider in this paper, although natural numbers and lists are incorporated in type theory there is no satisfactory mechanism for handling inductive data types in general.

Bengt Nordström has proposed an extension to type theory - the Acc type[4] - which provides a possible mechanism for dealing with the first two of these problems. The extension consists in adding a type of the form Acc(A,R). Something is a member of this type if it is a member of a set A well-ordered by the relation R. In this paper I will discuss a particular implementation of Martin-Löf's type theory, the Oyster system which has been developed in the Department of Artificial Intelligence at Edinburgh University, and I shall describe the mechanism for incorporating the Acc type in Oyster. There is also a discussion of a number of examples including a derivation of the quicksort program and the lambo program.

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks" grant 3245

2 Oyster

Oyster is an implementation of a system similar to Martin-Löfs constructive type theory carried out by C.Horn [3]. It is based on Cornell's NUPRL due to Constable *et al* [2]but is written in Prolog rather than Lisp, with only 4000 lines of code as against NUPRL's 60,000. It is a theorem proving system with an interactive proof editor based on a sequent calculus version of type theory. (Martin-Löf originally presented his type theory in natural deduction form). The implementation is due to Christian Horn.

In the Oyster implementation of constructive type theory, following NUPRL, we have the following types :

- Atom - the type atom (which models character strings),void (the empty type), pnat(natural numbers) and int(the integers). We also have three basic constructors: Cartesian product, disjoint union and function space. If A and B are types, then so is their Cartesian product A#B, their disjoint union A\B and the functions from A to B, A→B.
- The dependent product and dependent function constructors:

  x:A#B - there exists an x of type A such that B.

  x:A→B - for all x of type A, B is true.
- Lists, quotient and set types. The set type is written {x:A\B} and the quotient type allows Oyster to capture the idea of equivalence classes.
- recursive types and types of partial functions.
- Propositions considered as types.

The last type mentioned needs some explanation.

Consider first Martin-Löf's approach. For Martin-Löf there are four forms of judgment.

1. "A type" - A is a type
2. "A=B" - A and B are equal types
3. "a:A" - a is a member of a type
4. "a=b:A" - a and b are equal elements of type A.

The judgment that "a:A" can in fact be read in several different ways. Thus it may also be read as "a is a proof of proposition A", here we are regarding propositions as types with their proofs as members; or as "a is a program meeting specificaton A".

Martin-Löf draws a distinction between a judgment and a proposition. "a:A" is a *judgment* to the effect that a is a proof of the *proposition A"*.

Note that for Martin-Löf each type comes equipped with an equality relation. (Unlike classical set theory where equality is a general notion applicable to any set). In order to understand a type we must grasp the meaning of equality as defined over that type. One should also observe that while (4) is a judgment Martin-Löf finds it necessary to introduce a propositional form of equality.

Both Oyster and NUPRL are implementations of a sequent calculus, and are best understood as having only a single form of judgment written as follows:

$$[x_1 : A_1]......[x_n : A_n] \Longrightarrow G$$

The expression before the $\Longrightarrow$ is the context or environment; this is a sequence of declarations which can be taken as asserting that x_i is an object of type A_i or that x_i is a proof of A_i. G - the goal - may be the form "A ext a" where A is a proposition or type and a is an extract term, i.e. a computational object inhabiting that type, or it may be of the form "a=b in A". This latter is a proposition, not a judgment, and replaces Martin-Löf"s four types of judgment. "a in A" is construed as an abbreviation for "a=a in A". (1) and (2) are dealt with by postulating universes. In Oyster, following NUPRL, there is a chain of cumulative universes u(1), u(2), u(3)...to prevent the user running into Russell's paradox. u(1) consists of all small types i.e. the basic types - atom, pnat, void, int and all types which can be constructed from small types using the type constructors for list, union, product, function, set, quotient or recursive types. u(2) consists of all types of level (1) plus the universe u(1) itself and all types constructed from these again using all the type constructors. Judgment form (2) can then be replaced by "A=B in u(I)" for some I and "A type" by "A=A in u(I)".

Why is "a=a in A" regarded as a proposition in NUPRL? In Constable *et al*[2] it is pointed out that "a=b in A" makes sense only if A is a type and a and b are elements of that type. To express the fact that "a=b in A" is well-formed we show that it inhabits a universe. Thus we treat the well-formedness of such statements in the same way as the well-formedness of types and hence they must be regarded as propositions and not judgments. One should note that in Oyster as in NUPRL any identity can be regarded as proved by a canonical proof called "axiom". The reason for this is that we are not interested in the content of a proof of identity, it has no computational significance.

What of "A ext a" - this can be read "A is inhabited by extract term a". To prove a proposition in type theory is always to show that a proposition, construed as a type, is inhabited. We call the inhabitant of a type an extract term. This extract term is normally not displayed on the screen but at any stage in the proof development process it is possible to Access the extract term of the proof constructed thus far. Open sub-goals of the proof in so far as they have any computational significance and are not only well-formedness goals correspond to Prolog variables in the extract term. We can in fact argue that all goals are of the form "A ext a", if we are proving something of the form "a in A" we are in effect showing that "a in A" is inhabited by the extract term "axiom".

Associated with each type there is a selector construct together with a computation rule for its evaluation. Thus in the case of the Cartesian product we have:

$$spread(s, [l, r, t])$$

this evaluates to t with l and r replaced by the first and second components of s.

$$fst\ s{=}spread(s, [l, r, l])$$

$$snd\ s{=}spread(s, [l, r, r])$$

In the case of disjoint union we have that if a is in A, inl(a) is in A\B, and that if b is in B inr(B) is in A\B. We have:

$$decide(c, [u, e], [v, f])$$

if c is of the form inl(d) then this evaluates to the result of replacing u with d in e; and if c is of the form inr(d) then this evaluates to the result of replacing v with d in in f. So:

$$decide(inl(a), [u, e], [v, f]) = e[a/u]$$
$$decide(inr(b), [u, e], [v, f]) = f[b/v]$$

where x[y/z] is x with y replacing z.

3 Sub-types and Computational Redundancy

There are a number of differences between Martin-Löf's type theory and the Oyster implementation. One is that in Oyster as in NUPRL there is a notion of a sub-set which was absent from Martin-Löf's original theory. I shall briefly consider this notion.

One way of bringing out the connection between proofs and programs is to observe that in Martin-Löf's type theory the axiom of choice is provable. The axiom of choice is as follows:

$$\begin{aligned}&(n : pnat \to m : pnat \# R(n, m)) \to \\ &\qquad (f : (pnat \to pnat) \# n : pnat \to (R(n, f(n))))\end{aligned}$$

One can observe that the antecedent has the general form of a program specification. A specification states that for any input there is an output related to the input in the manner specified. The consequent of the axiom states that there is a function which applied to the input gives the output. We can think of the function as the program we want. As we are using a constructive logic then an existential claim must be backed by a witness, so if we then prove the consequent from the antecedent then the term inhabiting the conclusion would be that program. In fact however the extract term is a pair, the first component of which is indeed the the program we want and the second component is the extract term of a proof that this program satisfies the specification. This second component is, from the programmers view, of no interest. The trouble is that Martin-Löf's concept of a set is of a completely presented set, this means that instead of saying that we have evidence q that an item x is a member of a set X we say that the set X consists of pairs of which $< x, q >$ is an instance.

In practice it is the introduction rule for dependent products that causes the trouble. Essentially this is as follows. To prove a goal of the form

$$H \Longrightarrow (X : A \# B(X))$$

using the term F, we need to prove the subgoals:

$$\Longrightarrow F\ in\ A$$
$$\Longrightarrow B(F)\ ext\ G$$

together with a well-formedness goal. The extract term for the goal is F&G. The problem with this is that only F will be computationally interesting.

One method which will often get rid of computationally redundant information is to tag the computationally redundant information so that the extract term for our goal is F&G* (only G is tagged). At the end of a proof we can delete all tagged terms to obtain our program. Another device is to extend Martin-Löf's system with a notion of sub-sets. This is the method adopted in Oyster (also in NUPRL). Thus we write {X:A\B(X)} for the set of objects A having the property B. To prove a goal of the form H⟹{X:A\B(X)} we need to prove the subgoals:

$$\Longrightarrow F\ in\ A$$
$$\Longrightarrow B(F)$$

together with a well-formedness goal. The extract term for the goal is F. Thus the computationally uninteresting component of the proof is supressed.

However there are problems with the use of sub-sets. A well-known example concerns the root problem. We have a function which has the value 0 for some argument and we wish to compute that argument. In type theory the specification is:

$$(g : \{f : pnat \to pnat \backslash x : pnat\#f(x) = 0\})\{x : pnat \backslash g(x) = 0\}$$

i.e. (g:A)B(g).

If we could find an object b(g) which is a proof of B(g) on the assumption that g:A. Then the proof we want would be λg.b(g). b(g) is a proof of B(g) if b(g):pnat and we can find some proof, h say, of g(b(g))=0. If we can prove f(b(f))=0 on the assumption that f:(pnat→pnat) and that there is a proof j, say, of x:pnat#f(x)=0 then we can obtain our result by a rule analogous to existential elimination. This proof will be h* where h* is obtained from h by substituting f for g. If fst(j) is b(f) then h* is snd(j). But here we are stuck for how can we establish what j is? This information is precisely what we have throw away by using a sub-type in our specification of the type of g.

In general we need to restrict the elimination rule for sub-types. In Oyster Christian Horn has used the following method. To prove a goal of the form T(U) we need something of the form U:{V:A\B(V)} in the hypothesis list and we need to prove as a subgoal:

$$\begin{gathered} X : A \\ Y : B(U) \\ Z : X = U\ in\ A \Longrightarrow T(X)\ extract\ F \end{gathered}$$

as well as a well-formedness goal. The extract term as a result of the refinement step is F with X and Y replaced with U and assert(B(U) respectively. "assert" is in general not executable. So the extract will only be executable if Y does not occur free in F.

4 The Acc Type

Let us remind ouselves of the elimination rules for natural numbers:

$$\begin{gathered} a : pnat \\ n : C(0) \\ z(p, g) : C(succ(p))[p : pnat, g : C(p)] \\ \Longrightarrow rec(a, n, z) : C(a) \end{gathered}$$

(I put in square brackets the assumption that has to be made to derive the expression before the square brackets."succ" abbreviates "successor").

In course-of-values recursion the solution to the problem C(succ(x)) may depend on the solutions to C(0)...C(x). Nordström [4] suggests that a corresponding rule would be:

$$\begin{gathered} a : pnat \\ n : C(0) \\ z(p, g) : C(succ(p))[p : pnat, g(x) : C(x)[x \leq p)]] \\ \Longrightarrow covrec(a, n, z) : C(a) \end{gathered}$$

Nordström observes that we can simplify the above elimination rule to:

$$\begin{gathered} a : pnat \\ z(p, g) : C(p)[p : pnat, g(x) : C(x)[x < p)]] \\ \Longrightarrow covrec(a, z) : C(a) \end{gathered}$$

Nordström points out that there is nothing in the above rule that is particular to the natural numbers. The reason is the rule works is that pnat is well-ordered by $<$. The rule will generalise to an arbitrary set A which is well-ordered by some relation $<_A$. This gives us:

$$\begin{gathered} well - ordered(A, <_A) \\ a : A \\ z(p, g) : C(p)[p : A, g(x) : C(x)[x : A, x <_A p)]] \\ \Longrightarrow wo\_ind(a, z) : C(a) \end{gathered}$$

This is the principle of noetherian induction.

We will write $a : Acc(A, <_A)$ to mean that there are no infinite descending chains $... <_A a_2 <_A a_1 <_A a$. This gives us our introduction rule:

$$\begin{gathered} a : A \\ x : Acc(A, <_A)[x : A, x <_A < a] \end{gathered}$$

$$\Longrightarrow a \; in \; Acc(A, <_a)$$

The intuition here is that if for all x in A that precede a there are no infinite descending chains, starting from those x, then there are no infinite descending chains starting from a.

The elimination rule is:

$$\begin{array}{c} a : Acc(A, <_A) \\ z(p, g) : C(p)[p : A, g(x) : C(x)[x : A, x <_A < p]] \\ \Longrightarrow wo\_ind(a, z) \; in \; C(a) \end{array}$$

The computation rule is:

$$wo\_ind(a, z) = z(a, (x)wo\_ind(x, z)).$$

These rules have been implemented in a sequent formulation in Oyster.

The Acc type constitutes a possible solution to the two problems outlined earlier. Firstly, it constitutes a uniform mechanism for handling general recursion in type theory. Secondly, it provides a method for eliminating computationally redundant information. The reason for this is that a proof using that Acc type will consist of two parts; a proof that some relation is well-founded over some set and a proof of a goal using this as a lemma. The final extract term however will only contain computational information relating to the latter proof. The computational information relating to the former proof is 'thrown away' and the extract term is simply the program that we are attempting to construct.

5 Examples

5.1 Less than is well-ordered over the natural numbers

We prove that the < relation is well ordered over the natural numbers.

We need to prove $x : pnat => x \; in \; Acc(pnat, <)$. We use the introduction rule for the Acc type,this gives us a subgoals:

$$v1 : pnat, v1 < x \Longrightarrow v1 \; in \; Acc(pnat, <)$$

The proof is by induction on x. The base case is $v1 : pnat \to v1 < 0 \to v1 \; in \; Acc(pnat, <)$.We prove this by absurdity. The step case requires us to prove:

$$\begin{array}{l} v2 : pnat \\ v1 : pnat \to v1 < v2 \to v1 \; in \; Acc(pnat, <) \\ v3 : pnat \\ v3 < succ(v2) \\ \Longrightarrow v3 \; in \; Acc(pnat, <) \end{array}$$

We note that if $v4 : pnat$ and $v4 < v3$ and $v3 < succ(v2)$ then $v4 < v2$ and from the second of the above two assumptions we can deduce that $v4\ in\ Acc(pnat, <)$. Since we have $v3 < succ(v2)$ we can deduce that:

$$v4 : pnat, v4 < v3 \Longrightarrow v4\ in\ Acc(pnat, <)$$

But this is what we need to show that $v3\ in\ Acc(pnat, <)$.

5.2 Quicksort

5.2.1 Program Derivation

The type of lists is provided in Oyster. The canonical elements of the type A list are the empty list - nil - and the non-empty list $h :: t$. There is a list induction term $list\_ind(a, b, [h, t, v, f])$.

$$list\_ind(nil, b, [h, t, v, f]) = b$$
$$list\_ind(a :: r, b, [h, t, v, f]) = f[a, r, list\_ind(r, b, [h, t, v, f])/h, t, v]$$

The length of a list k is:

$$list\_ind(k, 0, [\_, \_, n, n+1])$$

If l is nil then this is 0, if k is h::t it is:

$$n + 1[h, t, list\_ind(t, 0, [\_, \_, n, n+1])/\ ,\ , n]$$

Before considering the program we will give a few definitions. We shall only consider lists of natural numbers.

$$len(l) \Longleftrightarrow list\_ind(l, 0, [h, t, v, s(v)]).$$

("len(l)" is the length of the list l).

$$occ\_in(x, l) \Longleftrightarrow list\_ind(l, 0, [h, t, v, pnat\_eq(x, h, s(v), v)]).$$

("occ_in(x,l)" is the number of occurrences of x in l).

$$perm(l, m) \Longleftrightarrow x : pnat \to (occ\_in(x, l) = occ\_in(x, m)\ in\ pnat \# len(l) = len(m) in pnat).$$

("perm(l,m)" means that list l is a permutation of list m).

$$true \Longleftrightarrow int.$$

$$false \Longleftrightarrow void.$$

$$ordered(l) \Longleftrightarrow list\_ind(l, true, [h, t, v, list\_ind(t, true,$$
$$[i, u, w, < (h, i, true, pnat\_eq(h, i, true, false))])\#v]).$$

("ordered(l)" means that list(l) is ordered).

$$listless \Longleftrightarrow \lambda l \lambda m.(len(l) < *len(m).$$

(listless of l of m means that the length of list l is less than the length of list m; we abbreviate "listless of l on m" to "listless(l,m)")

$$filter\_left(n,l) \Longleftrightarrow list\_ind(l, nil, [h,t,v,< (h,n,h :: v,v)]).$$
$$filter\_right(n,l) \Longleftrightarrow list\_ind(l, nil, [h,t,v,< (n,h,h :: v, pnat\_eq(n,h,h :: v,v))]).$$

("filter_left" and "filter_right" partition a list h::t into a list of elements of t less than h and into elements of t greater than or equal to t).

The specification for the quicksort program is:

$$m : pnat\ list \rightarrow \{l : pnat\ list \backslash perm(m,l)\#ordered(l)\}$$

We first prove as a lemma: do an intro and then sequence in:

$$\begin{aligned}&n : Acc(pnat\ list, \{listless\}) \rightarrow \\ &\qquad \{l : pnatlist \backslash perm(n,l)\#ordered(l)\}\end{aligned}$$

We then use the elimination rule for the Acc type which requires us to prove as a sub-goal

$$\begin{aligned}&\Longrightarrow v2 : pnat\ list \\ &v0 : (v1 : (\{v1 : pnat\ list \backslash listless(v1,v2)) \rightarrow \\ &\{l : pnat\ list \backslash perm(v1,l)\#ordered(l)\}) \\ &\Longrightarrow \{l : pnat\ list \backslash perm(v2,l)\#ordered(l)\}\end{aligned}$$

We prove by induction that given any list l then l is either nil or there is a natural number h and a list of natural numbers t such that l = h::t. We do a case case split. First we consider the v2 is nil. Clearly nil is an ordered permutation of nil so this case is trivial. In the other case we do an elimination on v0 with filter_left(h,t). This gives us as sub-goals:

$$\begin{aligned}&\Longrightarrow filter\_left(h,t)\ in\ \{v1 : (pnat\ list) \backslash listless(v1,v2) \\ &\{l : (pnat\ list) \backslash perm(filter\_left(h,t),l)\#ordered(l)\} \\ &\Longrightarrow \{l : pnat\ list \backslash perm(v2,l)\#ordered(l)\} by elim(v0, on(filter\_right(h,t)))\end{aligned}$$

We also do an elim on v0 with filter_right(h,t) and obtain similar sub-goals. We prove the first of each of the pairs of sub-goals by observing that the length of both the filters must be at least one less than the list to which they are applied. To prove the last of each of the pairs of sub-goals we observe that if we partition a list h::t with our filters and find well-ordered permutations of each list v (of filter_left(h,t)) and w (of filter_right(h,t)) then the result of appending v to h::w is an ordered permutation of our original list.

We have now proved that:

$$\begin{aligned}&n : Acc(pnat\ list, \{listless\}) \rightarrow \\ &\qquad \{l : pnatlist \backslash perm(n,l)\#ordered(l)\}\end{aligned}$$

To obtain our solution we need to show that $m\ in\ Acc(pnat, listless)$. We can do this by showing that lists are well-ordered by the listless relation. The proof is similar to that for the well-ordering of the natural numbers by $<$.

5.2.2 Evaluation

Consider now the evaluation of the quicksort program. In Oyster we write the selector for the acctype as:

$$wo\_ind(N, [X, Y, T])$$

We evaluate this as follows. Suppose that N evaluates to N*. Then the result of evaluating $wo\_ind(N, [X, Y, T])$ is T with X and Y replaced by N* and $\lambda V.(wo\_ind(V, [X, Y, T]))$ respectively.

The extract term for the quicksort program is as follows:

$$\begin{array}{l}\lambda m.(\lambda v0.(v0\ of\ m)\ of\\ \lambda n.(wo\_ind(n, [v2, v0, \lambda v1.(decide(v1\ of\\ v2, [\_, nil], [v5, spread(v5, [h, v4, spread(v4, [t, \_, app\ of\ (v0\ of\\ filter\_left(h, t))\ f\ (h :: v0\ of\ filter\_right(h, t))])])])])) \ of\\ lambda(l, list\_ind(l, inl(axiom),\\ [v1, v3, v4, inr(decide(v4, [\_, v1\&v3\&axiom], [\_, v1\&v3\&axiom]))])])])))\end{array}$$

Applied to a list of natural numbers it outputs an ordered permutation of that list.

5.3 Lambo

5.3.1 Program Derivation

Given a number m and a function $f : (pnat \to pnat)$ such that for some n, f of n is greater or equal to m *Lambo* $f\ m$ is the least n such that f of n is greater or equal than m. More formally the specification to find *lambo* $f\ m$ is

$$\begin{array}{l}m : pnat \to f : (\{v : (pnat \to pnat)\backslash n : pnat\#v\ of\ n < *m \to void\})\\ \to \{y : pnat\backslash t : pnat \to (t < *y \to f\ of\ t < *m)\#f\ of\ y < *m \to void\}\end{array}$$

We need to prove as a lemma that $Acc(A, succ)$, i.e. that this type is inhabited, where A is defined thus:

$$A = \{y : pnat\backslash t : pnat \to t < *y \to f\ of\ t < *m\}$$

We show this by showing that $0\ in\ Acc(A, succ)$. We use the Acc introduction rule which gives us as sub-goals:

$$\begin{array}{l}\Longrightarrow 0\ in\ A\\ v1 : A, v1 = succ(0) \Longrightarrow (v1\ in\ Acc(A, succ).).\end{array}$$

To show that 0 is in A we need to prove that 0 is in pnat, which is trivial, and that $t : pnat \to t < *0 \to f\ of\ t < *m$, which can be proved by absurdity. To show the second sub-goal let us suppose that $f\ of\ n < *m \to void$ so that we can then obtain our result by existential elimination. We now prove as a lemma:

$$t : pnat \to y : A \to (minus(n, y) = t\ in\ pnat) \to (y\ in\ Acc(A, succ).)$$

We prove this by an induction on t. The first sub-goal is the base case where n=y, so we need to show that $n\ in\ Acc(A, succ)$. Clearly $n : A$ so we need to prove

$$v2 : A, v2 = succ(y)\ in\ pnat \to (v2\ in\ Acc(A, succ).).$$

We prove this by absurdity as v2 cannot be in A if it is the successor of n.

We now deal with the step case of the induction. Given:

$$y : A \to minus(n, y) = v3\ in\ pnat \to y\ in\ Acc(A, succ).$$

we need to prove that:

$$y : A \to (minus(n, y) = succ(v3)\ in\ pnat) \to y\ in\ Acc(A, succ).$$

We use the Acc introduction rule which requires us to prove that:

$$v4 : A, succ(v4, y) \Longrightarrow v4\ in\ Acc(A, succ).$$

We eliminate the induction hypothesis on v4 to obtain

$$minus(n, v4) = v3\ in\ pnat \to v4\ in\ Acc(A, succ)$$

. Our result now follows by simple arithmetic.

It is easy to show that the sub-set A is well-ordered by the successor relation i.e. that $v1\ in\ Acc(A, succ)$ on the assumption that:

$$t : pnat \to y : A \to (minus(n, y) = t\ in\ pnat) \to (y\ in\ Acc(A, succ).)$$

Having shown that $Acc(A, succ)$. we need to show that if $v0 : Acc(A, succ)$. then the termination condition is satisfied. We use the Acc elimination rule which gives us as sub-goals:

$$v3 : A, v1 : (v2 : \{v2 : A \backslash v2 = succ(v3)\ in\ pnat\} \to P) \Longrightarrow P$$

Where P is the post condition of the program. We do a case analysis:

(1) Assume $f\ of\ v3 < m$.

We know that $v3 : A$ so we can do an elimination to obtain, $t : pnat \to t < v3 \to f\ of\ t < m$. We then do an $elim(v1, on(succ(v3)))$ which gives us as a first sub-goal to prove $succ(v3)\ in\ A$ and as a second sub-goal to prove that $succ(s(v3), v3)$ which is trivial. Taking our first sub-goal, this requires us to prove that $t : pnat \to t < succ(v3) \to f\ of\ t < m$. We can do this if we note that it follows from $t : pnat \to t < v3 \to\ f\ of\ t < m)$ and $f\ of\ v3 < m$, both of which we already have.

(2) Assume $f\ of\ v3 < m \to void$

Again we know that $v3 : A$ so we can do an elimination to obtain, $t : pnat \to t < v3 \to f\ of\ t < m$. We then do an intro on the postcondition which reqires us to prove that $t : pnat \to (t < v3 \to f\ of\ t < m)\#f\ of\ v3 < m \to void$. We already have both conjuncts.

5.3.2 Evaluation

The extract term for the lambo program is as follows:

$$\lambda m \lambda f \lambda v0.(wo\_ind(v0, [v2, v1,\\ pless(f\ of\ v2, m, v1\ of\ succ(v2), v2)])) \ of\ 0$$

Where pless(a,b,c,d) evaluates to c if a is less than b otherwise to d.

Applied to a number m and a function f this gives:

$$wo\_ind(0, [v2, v1, pless(f\ of\ v2, m, v1\ of\ succ(v2), v2)])$$

This gives:

$$pless(f\ of\ 0, m, \lambda v3.(wo\_ind(v3, [v2, v1,\\ pless(f\ of\ v2, m, v1 of\ succ(v2), v2)])) \ of\ succ(0), 0)$$

There are a number of further interesting examples of proofs involving the Acc type in Saaman and Malcolm [5].

6 Conclusion

The Oyster program can be used in order to conduct program synthesis. To synthesise a program one provides a logical specification decribing the inputs and outputs of the proposed program. Oyster can then be used as an interactive theorem prover to prove a conjecture to the effect that for any input there is an output related in the appropriate way to the input. As a further step one may seek to guide Oyster in its search for a proof. This procedure involves a number of difficulties. For example if one is concerned with inductive proofs one problem is to guide the choice of induction hypothesis. See Bundy *at al*[1].

A difficulty involved in such an enterprise is that even if one is able succesfully to guide the search for such a principle of induction then the proof one constructs will involve a proof of the correctness of this principle and the resulting extract term will contain superfluous information.

If one employs the Acctype then the problem for guiding Oyster in its search for a proof is not the choice of the principle of induction - this will uniformly be the principle of transfinite induction - but the choice of the appropriate well-founded relation. This means that if one successfully develops techniques for guiding the search for such a relation then the resulting proof will have two parts. The proof that the relation is indeed well-founded, and the proof of one's goal given that the relation in question is well-founded. The extract team one obtains will simply be the program one wants.

References

[1] A.Bundy, A.Smaill,J.Hesketh. "Turning Eurika Steps into Calculations in Automatic Program Synthesis" These proceedings.

[2] R.L.Constable *et al.* "Implementing Mathematics with the Nuprl Proof Development System." Prentice Hall, 1986.

[3] C.Horn. "The Nuprl Proof Development System." Working paper 214, Dept. of Artificial Intelligence,Edinburgh,1988. The Edinburgh version of Nuprl has been renamed Oyster.

[4] B. Nordström. "Terminating General Recursion." In preparation (1987).

[5] E.H.Saaman and G.R. Malcolm "Well-founded Recursion in Type Theory." Computing Science Notes 8710 Department of Mathematics and Computing Science of Groningen University

Implicit Syntax
DRAFT

Randy Pollack

LFCS, University of Edinburgh

Abstract

A proof checking system may support syntax that is more convenient for users than its 'official' language. For example LEGO (a typechecker for several systems related to the Calculus of Constructions) has algorithms to infer some polymorphic instantiations (e.g. `pair 2 true` instead of `pair nat bool 2 true`) and universe levels (e.g. `Type` instead of `Type(4)`). Users need to understand such features, but do not want to know the algorithms for computing them. In this note I explain these two features by non-deterministic operational semantics for "translating" implicit syntax to the fully explicit underlying formal system. The translations are sound and complete for the underlying type theory, and the algorithms (which I will not talk about) are sound (not necessarily complete) for the translations. This note is phrased in terms of a general class of type theories. The technique described has more general application.

1 Introduction

Consider the usual formal system $\lambda\!\rightarrow$ for simply typed λ-calculus (minimal implicational logic with explicit proofs):

$$\text{ENV} \qquad \Gamma \vdash x : \alpha \qquad\qquad x{:}\alpha \in \Gamma$$

$$\rightarrow\text{-I} \qquad \frac{\Gamma[x{:}\alpha] \vdash M : \beta}{\Gamma \vdash [x{:}\alpha]M : \alpha\rightarrow\beta} \qquad\qquad x \textit{ fresh}$$

$$\rightarrow\text{-E} \qquad \frac{\Gamma \vdash M : \alpha\rightarrow\beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash M\,N : \beta}$$

Suppose we have a proof-checker that decides if a given judgement is derivable. To keep the users happy, this proof checker allows proof terms with arbitrary elisions: any subterm may be replaced by a placeholder, say '?'. (Call such terms with elisions *implicit*.) The proof checker then decides whether or not the placeholders can be replaced with terms of the underlying system (explicit terms) such that the resulting term inhabits the claimed type. (Of course inhabitation (provability) in this system is decideable, so our mini-proofchecker can be complete, but in general such implicit syntax will involve incompleteness.) But what formal system does this proof checker really implement? Is it 'sound' with respect to $\lambda\!\rightarrow$? What does this question mean?

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

We could start talking about the algorithm: ...searching for derivations ...the search must terminate .... In order to claim that the implementation is sound and complete we *will* have to talk about the algorithm, but this gives a very complicated answer to the question of what formal system the proof checker implements. Nor is it a suitably abstract answer, since several different algorithms will implement this proof checker. Worse, the users may not be so happy after all when told that to know what their implicit proof terms mean they must understand some obscure algorithm.

In this note I suggest a way to explain such implicit syntax using non-deterministic operational semantics for "translating" from implicit syntax to explicit syntax. For example, for $\lambda{\rightarrow}$, which has a judgement of the shape $\Gamma \vdash M : \alpha$, we define a translation system with a judgement of shape $\Gamma, M, \alpha \Rightarrow (\Gamma \vdash M' : \alpha)$, meaning "$M'$ is an explicit form of M, and $\Gamma \vdash M' : \alpha$ is derivable in the underlying system, $\lambda{\rightarrow}$".

Notation 1 *Throughout this note I will informally use primes to indicate explicit syntactic objects; so χ' is an explicit form of χ. Whether or not we formally distinguish implicit and explicit as different syntactic classes, it is necessary to prove that the primed objects really are fully explicit.*[1]

To construct this translation system, consider those places in a derivation of $\lambda{\rightarrow}$ where ? may first occur. At such places non-deterministically translate the ? to any explicit term of the appropriate type. This suggests a translation axiom

$$\text{TRANS} \qquad \Gamma, ?, \alpha \Rightarrow (\Gamma \vdash M' : \alpha) \qquad\qquad \textit{if } \Gamma \vdash M' : \alpha$$

We also need rules saying that translation is a congruence for official derivations of $\lambda{\rightarrow}$.

$$\text{ENV} \qquad \Gamma, x, \alpha \Rightarrow (\Gamma \vdash x : \alpha) \qquad\qquad x{:}\alpha \in \Gamma$$

$$\rightarrow\text{-I} \qquad \frac{\Gamma[x{:}\alpha], M, \beta \Rightarrow (\Gamma[x{:}\alpha] \vdash M' : \beta)}{\Gamma, [x{:}\alpha]M, \alpha{\rightarrow}\beta \Rightarrow (\Gamma \vdash [x{:}\alpha]M' : \alpha{\rightarrow}\beta)} \qquad\qquad x \textit{ fresh}$$

$$\rightarrow\text{-E} \qquad \frac{\Gamma, M, \alpha{\rightarrow}\beta \Rightarrow (\Gamma \vdash M' : \alpha{\rightarrow}\beta) \qquad \Gamma, N, \alpha \Rightarrow (\Gamma \vdash N' : \alpha)}{\Gamma, M\,N, \beta \Rightarrow (\Gamma \vdash M'\,N' : \beta)}$$

In order to see what this translation system means in relation to $\lambda{\rightarrow}$, say that a term M in the extended language (with ?) is *explained* by a term M' in the official language of $\lambda{\rightarrow}$ (also written *M' explains M*) if M' is obtained by replacing each instance of ? in M by some term of $\lambda{\rightarrow}$. More exactly:

Definition 2 *? is* explained *by any term of $\lambda{\rightarrow}$. A variable x is* explained *by x.* Explanation *is extended compositionally to implicit terms.*

If $\Gamma, M, \alpha \Rightarrow (\Gamma \vdash M' : \alpha)$ is a translation judgement, we call $\Gamma \vdash M' : \alpha$ its right hand side (RHS). The essential soundness property of translation systems is that if a translation judgement is derivable, then its RHS is derivable in the underlying explicit system.

Theorem 3 (Soundness) *If $\Gamma, M, \alpha \Rightarrow (\Gamma \vdash M' : \alpha)$ then M' explains M and $\Gamma \vdash M' : \alpha$.*

[1]In this note the class of implicit objects is always an *extension* of the class of explicit objects, but this is not necessary; with translation we can forget syntax as well as add new syntax.

Proof By the usual induction: the axioms of the translation system establish the claimed property, and the rules of the translation system preserve it. In particular for both translation rules, the RHS of the premises over the RHS of the conclusion forms the corresponding rule in $\lambda\!\rightarrow$. ∎

1.0.2.1 Furthermore, this translation system justifies *every* suitable explanation of an implicit term.

Theorem 4 (Completeness) *If M' explains M and $\Gamma \vdash M' : \alpha$, then $\Gamma, M, \alpha \Rightarrow (\Gamma \vdash M' : \alpha)$.*

Proof Given a derivation, δ, of $\Gamma \vdash M' : \alpha$, and M explained by M', build a derivation of $\Gamma, M, A \Rightarrow (\Gamma \vdash M' : \alpha)$ following the shape of δ, and using δ to resolve any non-deterministic choices. ∎

Remark 5 *Derivations in the translation system above* almost *have the same shape as derivations of $\lambda\!\rightarrow$. This property is destroyed by the explicit reference to derivability of $\lambda\!\rightarrow$ in the side condition of translation axiom* TRANS. *This can be fixed by replacing* TRANS *with a rule that simply "reflects" derivability of $\lambda\!\rightarrow$ into derivability in the translation system:*

$$\text{TRANS} \qquad \frac{\Gamma, M, \alpha \Rightarrow (\Gamma \vdash M' : \alpha)}{\Gamma, ?, \alpha \Rightarrow (\Gamma \vdash M' : \alpha)}$$

See, for example, the translation rule Π-E *in Section 4.1*

1.0.2.2 In the remainder of this paper I give two substantial examples of translation over Type Systems. Section 2 gives a brief introduction to a general class of type systems in which the examples will be expressed. Section 3 presents a translation system to explain "typical ambiguity": making the levels in a type hierarchy implicit. An example of the use of typical ambiguity in Section 3.2 suggests that implicit syntax in not quite as straightfoward as it appears. In Section 4 implicit polymorphic instantiation (argument synthesis) is explained.

1.0.2.3 Acknowledgement Bob Harper pointed out to me the central idea that typical ambiguity may be explained by non-deterministically translating `Type` to *any* $\texttt{Type}_i$.

2 Type Systems

A very beautiful class of type systems has recently been studied. They are a generalization of the "Barendregt cube", introduced independently by S. Berardi [2] and J. Terlouw. Called *Generalized Type Systems* in [1, 4, 5, 9], Barendregt now suggests a change in terminology to *Pure Type Systems* (PTS). Their various extensions will be called *Type Systems* (TS). A PTS is essentially just "several copies of constructive universal-implicational logic hooked together". See [2, 1, 5] to get an idea how expressive this class is. We give a brief introduction to PTS and some TS.

2.1 Pure Type Systems

A Pure Type System (PTS) is a triple $(\mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule})$ with

- Sort, a set of *sorts*, (ranged over by s)
- $\mathsf{Ax} \subseteq \mathsf{Sort} \times \mathsf{Sort}$, a set of *axioms* of the form $(s_1 : s_2)$
- $\mathsf{Rule} \subseteq \mathsf{Sort} \times \mathsf{Sort} \times \mathsf{Sort}$, a set of *rules* of the form (s_1, s_2, s_3) (written (s_1, s_2) when $s_3 = s_2$)

2.1.0.4 Syntax Let x range over an infinite set of variables. The raw syntax of *terms*, *contexts* and *judgements* of PTS $P = (\mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule})$ are given by:

$$
\begin{array}{llcll}
\textbf{terms} & M & ::= & x & \textit{variable} \\
 & & \mid & s & \textit{sort} \\
 & & \mid & [x{:}M]M & \textit{lambda} \\
 & & \mid & \{x{:}M\}M & \textit{Pi} \\
 & & \mid & M\,M & \textit{application} \\
\textbf{contexts} & \Gamma & ::= & \bullet & \textit{empty} \\
 & & \mid & \Gamma[x{:}\,M] & \\
\textbf{judgement} & J & ::= & \Gamma \vdash M : M &
\end{array}
$$

Call the set of raw terms of a PTS a *pure language*. Substitution, β-reduction, and β-conversion are defined as usual for pure languages. β-conversion is Church-Rosser.

2.1.0.5 Typing The judgements of P are those derivable from the axioms and inference rules of Table 1. As usual we abuse notation by writing $\Gamma \vdash M : M$ to mean the judgement is derivable. The basic meta-theoretic properties of PTS are presented in [2, 4, 9].

2.1.0.6 Examples The Pure Calculus of Constructions [3] can be presented as the PTS $\lambda P\omega$:

$$
\lambda P\omega \qquad
\begin{array}{rl}
\mathsf{Sort} & *, \Box \\
\mathsf{Ax} & (* : \Box) \\
\mathsf{Rule} & (*, *), (*, \Box), (\Box, *), (\Box, \Box)
\end{array}
$$

The Edinburgh Logical Framework [6] can be presented as the PTS λP:

$$
\lambda P \qquad
\begin{array}{rl}
\mathsf{Sort} & *, \Box \\
\mathsf{Ax} & (* : \Box) \\
\mathsf{Rule} & (*, *), (*, \Box)
\end{array}
$$

To understand these presentations see [2, 1].

AX	$\bullet \vdash s_1 : s_2$	$(s_1 : s_2) \in \mathsf{Ax}$
START	$\dfrac{\Gamma \vdash A : s}{\Gamma[x{:}A] \vdash x : A}$	x *fresh*
WEAK	$\dfrac{\Gamma \vdash B : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash B : C}$	x *fresh*
Π-F	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma[x{:}A] \vdash B : s_2}{\Gamma \vdash \{x{:}A\}B : s_3}$	$(s_1, s_2, s_3) \in \mathsf{Rule}$
Π-I	$\dfrac{\Gamma[x{:}A] \vdash M : B \qquad \Gamma \vdash \{x{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$	
Π-E	$\dfrac{\Gamma \vdash M : \{x{:}A\}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : [N/x]B}$	
CONV	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \simeq B}{\Gamma \vdash M : B}$	

Table 1: The Typing Judgement of a PTS

2.2 Some Extensions: Type Systems

2.2.1 TS with Sigma Types

To TS $(\mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule}, \ldots)$ add a new set of rules controlling Σ-Formation

$$\Sigma\mathsf{Rule} \subseteq \mathsf{Sort} \times \mathsf{Sort} \times \mathsf{Sort}$$

and typing rules

$$\Sigma\text{-F} \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma[x{:}A] \vdash B : s_2}{\Gamma \vdash \Sigma x{:}A.B : s_3} \qquad (s_1, s_2, s_3) \in \Sigma\mathsf{Rule}$$

$$\Sigma\text{-I} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : [M/x]B \qquad \Gamma \vdash \Sigma x{:}A.B : s}{\Gamma \vdash (M, N)_{\Sigma x:A.B} : \Sigma x{:}A.B}$$

$$\pi 1 \qquad \frac{\Gamma \vdash M : \Sigma x{:}A.B}{\Gamma \vdash M.1 : A}$$

$$\pi 2 \qquad \frac{\Gamma \vdash M : \Sigma x{:}A.B}{\Gamma \vdash M.2 : [M.1/x]B}$$

Notice that dependent pairs carry explicit type information.

2.2.2 TS with Cumulativity

To TS $(\mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule}, \ldots)$ add a relation

$$\mathsf{Cum} \subseteq \mathsf{Sort} \times \mathsf{Sort}$$

Definition 6 *$\preceq$ is the smallest partial order (over $\simeq$-classes of terms) such that*

- $\mathsf{Cum} \subseteq \preceq$
- *if $A' \preceq A$ and $B \preceq B'$ then $\{x{:}A\}B \preceq \{x{:}A'\}B'$*
- *if $A \preceq A'$ and $B \preceq B'$ then $\Sigma x{:}A.B \preceq \Sigma x{:}A'.B'$*

To the typing rules of the TS, add the rule

$$\text{CUM} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \preceq B}{\Gamma \vdash M : B}$$

Luo's ECC [11, 15] is an instance of TS with Sigma and Cumulativity.

3 Typical Ambiguity

Suppose $\Box_i$ ($i \in \omega$) are among the sorts of a TS $P = (\mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule}, \ldots)$, and $\Box_i : \Box_{i+1}$ ($i \in \omega$) are among it's axioms. In ECC [11, 15], for example, there are different identity functions polymorphic over each $\Box_i$:

$$\begin{array}{c} I_0 \equiv [T{:}\Box_0][x{:}T]x : \{T{:}\Box_0\}T \rightarrow T \\ I_1 \equiv [T{:}\Box_1][x{:}T]x : \{T{:}\Box_1\}T \rightarrow T \\ \vdots \end{array}$$

We introduce implicit syntax $\Box$, standing for "any $\Box_i$", such that a single identity function can be defined for all $\Box_i$

$$I \equiv [T{:}\Box][x{:}T]x : \{T{:}\Box\}T \rightarrow T$$

This syntax is explained by a translation into the 'official' TS.

3.1 The 'Typical Ambiguity' Translation

Extend the language of P:

$$M \ ::= \ \ldots \ \mid \ \Box$$

This extended language is still a pure language. As before let s, s_1, ..., range over Sort, the sorts of P, which does not include the new symbol $\Box$. We define a formal system for translation, with judgements of the form $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$ meaning "Γ', M' and A' are explicit forms of Γ, M and A, and $\Gamma' \vdash M' : A'$". This formal system is *not* itself a TS.

In order to do the translation, consider the places where some $\Box_i$ might first appear in some derivation of P. From the axioms of P we get the translation axioms:

$$\text{AX} \qquad \bullet, \Box, \Box \Rightarrow (\bullet \vdash \Box_i : \Box_j) \qquad (\Box_i{:}\Box_j) \in \mathsf{Ax}$$

$$\text{AX} \qquad \bullet, \Box, s \Rightarrow (\bullet \vdash \Box_i : s) \qquad (\Box_i{:}s) \in \mathsf{Ax}$$

$$\text{AX} \qquad \bullet, s, \Box \Rightarrow (\bullet \vdash s : \Box_j) \qquad (s{:}\Box_j) \in \mathsf{Ax}$$

That is we non-deterministically translate each occurrence of $\Box$ to some $\Box_i$, *respecting the axioms* of the underlying TS.

Since a sort may occur in the conclusion of rule Π-F without occurring in its premises, the translation system must have a rule:

$$\Pi\text{-F} \qquad \frac{\Gamma, A, s_1 \Rightarrow (\Gamma' \vdash A' : s_1) \qquad \Gamma[x{:}A], B, s_2 \Rightarrow (\Gamma'[x{:}A'] \vdash B' : s_2)}{\Gamma, \{x{:}A\}B, \Box \Rightarrow (\Gamma' \vdash \{x{:}A'\}B' : \Box_i)} \qquad (s_1, s_2, \Box_i) \in \mathsf{Rule}$$

In this rule, $\Box$ is translated non-deterministically to some $\Box_i$, but "coherence of translation" is enforced between the two premises: Γ is translated to the same Γ' in both premises.

All other occurrences of a sort in the conclusions of rules are residuals of occurrences in their premises; we need rules expressing that translation is a congruence for official derivations. These congruence rules are collected in Table 2. Notice that the rules of Table 2 are not specific to the typical ambiguity translation, but occur in all systems for translation into a PTS.

AX	$\bullet, s_1, s_2 \Rightarrow (\bullet \vdash s_1 : s_2)$	$(s_1{:}s_2) \in \mathsf{Ax}$
START	$\dfrac{\Gamma, A, s \Rightarrow (\Gamma' \vdash A' : s)}{\Gamma[x{:}A], x, A \Rightarrow (\Gamma'[x{:}A'] \vdash x : A')}$	x *fresh*
WEAK	$\dfrac{\Gamma, B, C \Rightarrow (\Gamma' \vdash B' : C') \qquad \Gamma, A, s \Rightarrow (\Gamma' \vdash A' : s)}{\Gamma[x{:}A], B, C \Rightarrow (\Gamma'[x{:}A'] \vdash B' : C')}$	x *fresh*
Π-F	$\dfrac{\Gamma, A, s_1 \Rightarrow (\Gamma' \vdash A' : s_1) \qquad \Gamma[x{:}A], B, s_2 \Rightarrow (\Gamma'[x{:}A'] \vdash B' : s_2)}{\Gamma, \{x{:}A\}B, s_3 \Rightarrow (\Gamma' \vdash \{x{:}A'\}B' : s_3)}$	$(s_1, s_2, s_3) \in \mathsf{Rule}$
Π-I	$\dfrac{\Gamma[x{:}A], M, B \Rightarrow (\Gamma'[x{:}A'] \vdash M' : B') \qquad \Gamma, \{x{:}A\}B, s \Rightarrow (\Gamma' \vdash \{x{:}A'\}B' : s)}{\Gamma, [x{:}A]M, \{x{:}A\}B \Rightarrow (\Gamma' \vdash [x{:}A']M' : \{x{:}A'\}B')}$	
Π-E	$\dfrac{\Gamma, M, \{x{:}A\}B \Rightarrow (\Gamma' \vdash M' : \{x{:}A'\}B') \qquad \Gamma, N, A \Rightarrow (\Gamma' \vdash N' : A')}{\Gamma, M\,N, [N/x]B \Rightarrow (\Gamma' \vdash M'\,N' : [N'/x]B')}$	
CONV	$\dfrac{\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A') \qquad \Gamma, B, s \Rightarrow (\Gamma' \vdash B' : s) \qquad A' \simeq B'}{\Gamma, M, B \Rightarrow (\Gamma' \vdash M' : B')}$	

Table 2: Translation is a congruence for official PTS derivations

3.1.1 Soundness and Completeness

Definition 7 *$\Box$ is* explained *by any $\Box_i$. s is* explained *by s. x is* explained *by x.* Explanation *is extended compositionally to terms.*

Theorem 8 (Soundness) *If $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$ then Γ', M', A' explain Γ, M, A respectively, and $\Gamma' \vdash M' : A'$.*

Theorem 9 (Completeness) *If Γ', M', A' explain Γ, M, A respectively, and $\Gamma' \vdash M' : A'$, then $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$.*

The proofs are as in the example of Section 1.

3.1.2 Implementation

Notice that Theorems 8 and 9 claim only that the translation relation is related to the typing relation of P. In fact, if the well-typed terms of P are (weakly) normalizing, then both the typing relation, and the translation relation are decidable [7, 8]

3.2 An Example

Consider the TS I will call λPU, "λP extended with stratified, cumulative universes":

λPU	Sort	$\Box_i$	$(i \in \omega)$
	Ax	$(\Box_i : \Box_{i+1})$	$(i \in \omega)$
	Rule	$(\Box_i, \Box_j, \Box_{\max(i,j)})$	$(i, j \in \omega)$
	Cum	$(\Box_i, \Box_{i+1})$	$(i \in \omega)$

Note that λPU is completely predicative. The well-typed terms of λPU are strongly normalizing [15], so type checking and type synthesis are decidable for this system, and for its associated typical ambiguity translation system.

With $I \equiv [A{:}\Box][x{:}A]x$ and $I_j \equiv [T{:}\Box_j][x{:}T]x$, have

$$\bullet, I, \{T{:}\Box\}T \to T \Rightarrow (\bullet \vdash I_j : \{T{:}\Box_j\}T \to T) \qquad (j \in \omega)$$

and[2]

$$\bullet, I \,?\, I, \{T{:}\Box\}T \to T \Rightarrow (\bullet \vdash I_{j+i} \,?\, I_j : \{T{:}\Box_j\}T \to T) \qquad (j, i \in \omega)$$

Notice that different occurrences of I may get different translations.

To see that this system doesn't have the infamous properties of $\Box : \Box$ notice that the arbitrary object whose type is that of I cannot be self applied[3]:

$$[J{:}\{A{:}\Box\}A \to A], J \,?\, J, X \not\Rightarrow \Gamma \vdash Y : Z \qquad (\textit{all } X, \Gamma, Y, Z)$$

[2]To clarify this example I will elide explicit polymorphism to ?. Such elision is explained in Section 4

[3]This example is from [8]

In λPU the "polymorphic" Church numerals are definable

$$nat \equiv \{A{:}\Box\}A{\rightarrow}(A{\rightarrow}A){\rightarrow}A$$

along with their zero, successor, addition, multiplication, ... How far can this be carried? See [10]. Similarly some "impredicative" logic can be carried out in λPU

$$and \equiv [A,B{:}\Box]\{C{:}\Box\}(A{\rightarrow}B{\rightarrow}C){\rightarrow}C$$
$$\vdots$$

How much logic can be done in this system?

4 Argument Synthesis

4.1 Ad Hoc Argument Synthesis

LEGO can infer some arguments of applications. Extend the syntax by

$$M \ ::= \ \ldots \ \mid \ ?$$

Over $\lambda P\omega$ (Pure Constructions), for example, one can write `pair ? ? 3 true` instead of `pair nat bool 3 true`, since the types of the arguments `3` and `true` can be inferred. This is very similar to "partial type inference" in $F\omega$ [13], except that in general a TS may have different dependencies than $F\omega$, so entities other than "types" can be inferred. For example, in λP (the Edinburgh Logical Framework) and $\lambda P\omega$ types may depend on elements (proofs), so elements can be inferred as well as types.

To explain this syntax, we present a translation system. Let $P = (\mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule}, \ldots)$, be a TS, and consider where the new syntax ? may first occur in a derivation. At such places, translate ? non-deterministically to any term consistent with the underlying TS, P. There is only one interesting translation rule, which comes from the application rule of P

$$\Pi\text{-E} \qquad \frac{\Gamma, M, \{x{:}A\}B \Rightarrow (\Gamma' \vdash M' : \{x{:}A'\}B') \qquad \Gamma, N, A \Rightarrow (\Gamma' \vdash N' : A')}{\Gamma, M\,?\,, [N/x]B \Rightarrow (\Gamma' \vdash M'\,N' : [N'/x]B')}$$

The translation system also has all the rules of Table 2 saying that translation is a congruence for official derivations.

For example, extending $\lambda P\omega$ with ? and taking $I \equiv [A{:}*][x{:}A]x$, we have

$$\bullet, I\,?\,I, \{A{:}*\}A{\rightarrow}A \Rightarrow (\bullet \vdash I\,(\{A{:}*\}A{\rightarrow}A)\,I : \{A{:}*\}A{\rightarrow}A)$$

This explains the elision used in the examples of Section 3.2.

Definition 10 ? *is* explained *by any explicit term.* s *is* explained *by* s. x *is* explained *by* x. Explanation *is extended compositionally to terms, contexts and judgements.*

Theorem 11 (Soundness) *If* $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$ *then* Γ', M', A' *explain* Γ, M, A *respectively, and* $\Gamma' \vdash M' : A'$.

Theorem 12 (Completeness) *If* Γ', M', A' *explain* Γ, M, A *respectively, and* $\Gamma' \vdash M' : A'$, *then* $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$.

The proofs are as above.

4.1.1 Implementation

First notice that it is the possible occurrence of N' as a subterm of $[N'/x]B'$ that makes translation rule Π-E partially implementable.

Although this translation system has the same soundness and completeness *with respect to the underlying TS* as does the translation for typical ambiguity above, we cannot hope to have a complete implementation in the present case. (As Rod Burstall observed "addition can't infer it's arguments!") It is *soundness* of implementations that we really want to prove. Theorem 11 explains what we mean by "a sound implementation of argument synthesis", namely a sound implementation of the translation system above.

4.2 Uniform Argument Synthesis

We further extend the system of Section 4.1 with syntax to say uniformly which arguments are to be inferred. This eliminates the need to include the placeholder ? at all. Extend the syntax of Section 4.1 by

$$M \ ::= \ \ldots \ \mid \ [x|M]M \ \mid \ \{x|M\}M \ \mid \ M|M$$

Call these new term constructs *bar-lambda*, *bar-pi* and *bar-application* respectively.

For example, if we define $I \equiv [A|*][x{:}A]x$, the type inferred for I will be $\{A|*\}A{\rightarrow}A$. This shows that I is a function of 2 arguments, the first of which will be inferred. Now $I\,I$ is well-typed. However $I\ nat$ is no longer well-typed; to allow this specialization of polymorphic functions, there is a new form of application to "override" implicit argument synthesis. Thus $I|nat$ has type $nat{\rightarrow}nat$. To explain this syntax, we extend the translation system of Section 4.1. First there are rules that translate the new binders $[x|M]M$ and $\{x|M\}M$ into their 'un-barred' counterparts:

$$\Pi\text{-F} \qquad \frac{\Gamma, A, s_1 \Rightarrow (\Gamma' \vdash A' : s_1) \qquad \Gamma[x{:}A], B, s_2 \Rightarrow (\Gamma'[x{:}A'] \vdash B' : s_2)}{\Gamma, \{x|A\}B, s_3 \Rightarrow (\Gamma' \vdash \{x{:}A'\}B' : s_3)} \qquad (s_1, s_2, s_3) \in \mathsf{Rule}$$

$$\Pi\text{-I} \qquad \frac{\Gamma[x{:}A], M, B \Rightarrow (\Gamma'[x{:}A'] \vdash M' : B') \qquad \Gamma, \{x{:}A\}B, s \Rightarrow (\Gamma' \vdash \{x{:}A'\}B' : s)}{\Gamma, [x|A]M, \{x|A\}B \Rightarrow (\Gamma' \vdash [x{:}A']M' : \{x{:}A'\}B')}$$

Intuitively, the only purpose of the | in a bar-lambda is to mark that its type is a bar-pi. To see the purpose of bar-pi we look at the Π-elimination rule. Besides the usual Π-elimination rule (Table 2) we have two new translation rules explaining elimination of bar-pi:

$$\Pi\text{-E} \qquad \frac{\Gamma, M, \{x|A\}B \Rightarrow (\Gamma' \vdash M' : \{x{:}A'\}B') \qquad \Gamma, N, A \Rightarrow (\Gamma' \vdash N' : A')}{\Gamma, M|N, [N/x]B \Rightarrow (\Gamma' \vdash M'\,N' : [N'/x]B')}$$

$$\Pi\text{-E'} \qquad \frac{\Gamma, M, \{x|A\}B \Rightarrow (\Gamma' \vdash M' : \{x{:}A'\}B') \qquad \Gamma, M'\,?\,N, C \Rightarrow (\Gamma' \vdash Q' : C')}{\Gamma, M\,N, C \Rightarrow (\Gamma' \vdash Q' : C')}$$

The first of these just says 'bar-application eliminates bar-pi'. The second is the interesting rule, saying that un-barred application can *trigger* the elimination of bar-pi by a new inferred argument.

We now have rules for three cases: elimination of bar-pi by bar-application and by application, and elimination of pi by application (in Table 2). Of course there is a fourth case, elimination of pi by bar-application; we can be liberal or strict by including or rejecting the obvious rule for this case.

Remark 13 *As an example of the usefulness of this style of explanation, consider an alternative to the last rule,* Π-E'*:*

$$\text{ELF} \qquad \frac{\Gamma, M, \{x|A\}B \Rightarrow (\Gamma' \vdash M' : \{x{:}A'\}B') \qquad \Gamma, M'\,?, C \Rightarrow (\Gamma' \vdash Q' : C')}{\Gamma, M, C \Rightarrow (\Gamma' \vdash Q' : C')}$$

(I believe rule ELF *is close to the choice made in the logic programming language ELF [14]. However ELF uses different syntax to annotate implicit arguments.) This rule is more eager in its elimination of bar-pi than is rule* Π-E'*. Notice that an (implicit) term may have both functional and non-functional type using rule* ELF*, but this anomaly can be corrected with an auxiliary judgement. LEGO uses rule* Π-E' *to avoid both the anomaly and the need for an auxiliary judgement: a function has functional type until it is applied, even if some of its arguments are implicit. Rule* ELF*, however, is 'stronger' in that it allows more arguments to be inferred. For example in LEGO the polymorphic* nil *cannot infer its one argument because there is no second argument to 'trigger' the inference, while rule* ELF *(and the language ELF) allows such inference.*

Definition 14

1. *? is* explained *by any explicit term.*
2. *s is* explained *by s.*
3. *x is* explained *by x.*

If M, N, A and B are explained by M', N', A' and B', then

4. *$\{x|A\}B$ and $\{x{:}A\}B$ are* explained *by $\{x{:}A'\}B'$.*
5. *$[x|A]M$ and $[x{:}A]M$ are* explained *by $[x{:}A']M'$*
6. *$M|N$ is* explained *by $M'\,N'$*
7. *$M\,N$ is* explained *by $M'\,N'$ and by $M'\,X\,N'$, where X is any explicit term.*

Theorem 15 (Soundness) *If $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$ then Γ', M', A' explain Γ, M, A respectively, and $\Gamma' \vdash M' : A'$.*

Theorem 16 (Completeness) *If Γ', M', A' explain Γ, M, A respectively, and $\Gamma' \vdash M' : A'$, then $\Gamma, M, A \Rightarrow (\Gamma' \vdash M' : A')$.*

References

[1] H. Barendregt. "Introduction to Generalised Type Systems", in *Proceedings of the Third Italian Conference on Theoretical Computer Science*, Ed. U. Moscati, World Scientific Publishing Co., Singapore (1989).

[2] S. Berardi. *Type Dependence and Constructive Mathematics.* Ph.D. thesis, Dipartimento di Informatica, Torino, Italy (1990).

[3] Th. Coquand and G. Huet. "The Calculus of Constructions", *Information and Computation* 76,95-120 (1988).

[4] H. Geuvers and M.-J. Nederhof. *A Modular Proof of Strong Normalization for the Calculus of Constructions.* Faculty of Mathematics and Informatics, Catholic University Nijmegen (draft, June 1989).

[5] H. Geuvers. *Type Systems for Higher Order Logic.* Faculty of Mathematics and Informatics, Catholic University Nijmegen (draft, April 1990).

[6] R. Harper, F. Honsell and G. Plotkin. "A Framework for Defining Logics", in *Proceedings Symposium on Logic in Computer Science*, Ithaca, New York (June 1987).

[7] R. Harper and R. Pollack. "Typechecking, Universe Polymorphism, and Typical Ambiguity", in *Proceedings, TAPSOFT '89*, Eds. J. Diaz, F. Orejas, LNCS 352 (1989).

[8] G. Huet. *Extending the Calculus of Constructions with Type:Type.* Undistributed draft (April 1987).

[9] L.S. van Bentham Jutting. *Typing in Generalized Type Systems.* Unpublished note presented at IV Jumelage meeting, Bari, Italy (May 1990).

[10] D. Leivant. "Stratified Polymorphism", in *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, Asilomar, California (June 1989).

[11] Z. Luo. "ECC, an Extendedf Calculus of Constructions", in *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, Asilomar, California (June 1989).

[12] Z. Luo *An Extended Calculus of Constructions.* Ph.D. Thesis, Department of Computer Science, University of Edinburgh (June 1990).

[13] F. Pfenning. "Partial Polymorphic Type Inference and Higher-Order Unification", in *Proceedings 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah (July 1988).

[14] F. Pfenning. "Elf: A Language for Logic Definition and Verified Mataprogramming", in *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, Asilomar, California (June 1989).

INVESTIGATIONS INTO PROOF-SEARCH IN A SYSTEM OF FIRST-ORDER DEPENDENT FUNCTION TYPES

David Pym
University of Edinburgh
Scotland, U.K.
dpym@uk.ac.ed.lfcs

Lincoln Wallen
University of Oxford
England, U.K.
lw@uk.ac.ox.prg

Abstract

We present a series of proof systems for $\lambda\Pi$-calculus: a theory of *first-order dependent function types*. The systems are complete for the judgement of interest but differ substantially as bases for algorithmic proof-search. Each calculus in the series induces a search space that is properly contained within that of its predecessor. The $\lambda\Pi$-calculus is a candidate *general logic* in that it provides a metalanguage suitable for the encoding of logical systems and mathematics. Proof procedures formulated for the metalanguage extend to suitably encoded object logics, thus removing the need to develop procedures for each logic independently. This work is also an exploration of a *systematic* approach to the design of proof procedures. It is our contention that the task of designing a computationally efficient proof procedure for a given logic can be approached by formulating a series of calculi that possess specific proof-theoretic properties. These properties indicate that standard computational techniques such as *unification* are applicable, sometimes in novel ways. The study below is an application of this design method to an intuitionistic type theory.

Our methods exploit certain forms of *subformula property* and *reduction ordering* — a notion introduced by Bibel for classical logic, and extended by Wallen to various non-classical logics — to obtain a search calculus for which we are able to define notions of *compatibility* and *intrinsic well-typing* between a derivation ψ and a substitution σ calculated by unification which closes ψ (a derivation is closed when all of its leaves are axioms). Compatibility is an acyclicity test, a generalization of the *occurs-check* which, subject to intrinsic well-typing, determines whether the derivation ψ and substitution σ together constitute a *proof.*

Our work yields the (operational) foundations for a study of *logic programming* in this general setting. This potential is not explored here.

1 Introduction

We present a series of proof systems for $\lambda\Pi$-calculus: a theory of *first-order dependent function types* [vDa80,MR86,HHP87]. The systems are complete for the judgement of interest but differ substantially as bases for algorithmic proof-search. Each calculus in the series induces a search space that is properly contained within that of its predecessor.

Our interest in proof-search in $\lambda\Pi$-calculus stems from two sources. Firstly, the theory is a candidate *general logic* in that it provides a metalanguage suitable for the encoding of logical systems [AHM87] and mathematics [vDa80]. Proof procedures formulated for the metalanguage extend to suitably encoded object logics, thus removing the need to develop procedures for each logic independently.[1] The second reason for our interest arises out from a desire (expressed in [Wal89]) for a *systematic* approach to the design of proof procedures. It is our contention that the task of designing a computationally efficient proof procedure for a given logic can be approached by formulating a series of calculi that possess specific proof-theoretic properties. These properties are the indicators that standard computational techniques such as *unification* are applicable, sometimes in novel ways. The study below is an application of this design method to an intuitionistic type theory.

The first system of the series, called **N**, is a natural deduction formulation of $\lambda\Pi$-calculus.[2] The main judgement of **N** is the typing assertion: $\Gamma \vdash M{:}A$, meaning that the term M has type A, given the type assignments for free variables and constants recorded in Γ. This relation is decidable [HHP87]. A typical rule of **N** is the elimination rule for the dependent function type constructor (Π):[3]

$$\Pi\mathcal{E}\,\frac{\Gamma \vdash M{:}\,(\Pi x{:}A.B) \qquad \Gamma \vdash N{:}\,A}{\Gamma \vdash MN{:}\,B[N/x]}$$

where $B[N/x]$ denotes capture-avoiding substitution of N for free occurrences of x in B.

The second system, called **L**, is sound and complete (relative to **N**) for the semi-decidable relation of *inhabitation:* $\Gamma \Rightarrow A$, with the meaning $(\exists M)(\Gamma \vdash M{:}A)$. The judgements of **L** assert the *existence* of proofs of the judgements of **N**, as in the case of first-order logic [Pra65]. **L** is the starting point for our investigation into proof-search. A typical rule of **L** is the Π-left rule, the counterpart of the Π-elimination rule of **N**:

$$\Pi l\,\frac{\Gamma, z{:}B[N/x] \Rightarrow C}{\Gamma \Rightarrow C} \qquad \begin{array}{l}\text{(a) } w : (\Pi x{:}A.B) \in \Gamma \\ \text{(b) } z \notin \mathrm{Dom}(\Gamma) \\ \text{(c) } \Gamma \vdash N{:}A.\end{array}$$

L is almost a *logicistic* system, in the sense of Gentzen, meaning that there is only one localized appeal to an external notion, that notion being an appeal to the system **N** in side condition (c) of the rule above. Indeed, with respect to the Π-type structure of terms in the language, **L** has a *subformula property* [Gen34]. As a consequence, if the inference rules are used as *reduction operators* from conclusion to premisses then **L** induces a search space of derivations of a given sequent.[4] Notice that if the Πl rule is used as a reduction, the choice of term N to use in the premiss is unconstrained by the conclusion of the rule. The sub*formula* property of the Π-types does not extend to a full sub*term* property (*cf.* the quantifier rules of the predicate calculus). The *axiom* sequent (or *closure* condition for the reduction system) is:

$$\Gamma, z{:}A, \Gamma' \Rightarrow A$$

i.e., the conclusion occurs as the type of a declaration in the context.

[1] This should be compared with the use of proof procedures for classical logic to effect proof-search within a modal logic, say, by means of an encoding of the latter in the former; see [Ohl88] for example.

[2] This system is also known as the type system of the (Edinburgh) Logical Framework or LF [HHP87].

[3] Readers that are unfamiliar with dependent types should read the construction $\Pi x{:}A.B$ as a (typed) universal quantifier such as $\forall x{:}A.B$; x is the bound variable, ranging over the type A, which may occur free in the term B.

[4] Kleene [Kle68] explains this in the case of the predicate calculus. Sequent systems used in this way are systems of *block tableaux* [Smu68].

The third and fourth systems are also systems of sequents. The system **U** is formed from **L** by removing the appeal to **N** in the Πl rule. The Πl rule of **U** is thus:

$$\Pi l \frac{\Gamma, z{:}B[\alpha/x] \Rightarrow C}{\Gamma \Rightarrow C} \quad \begin{array}{l} \text{(a) } w : (\Pi x{:}A.B) \in \Gamma \\ \text{(b) } z \notin \mathrm{Dom}(\Gamma). \end{array}$$

This rule introduces a free, or *universal* variable into the proof, denoted here by α. Universal variables are distinct from the usual *eigenvariables* that are bound by the derivation (and explicitly declared in contexts). The axiom sequent of **U** is used to compensate for the omission of term information in the Πl rule as follows:

$$\Gamma, z{:}B, \Gamma' \Rightarrow A \quad \text{(a) } (\exists\sigma)\ B\sigma \equiv A\sigma$$

where σ is an *instantiation* of the universal variables of the sequent, and $B\sigma$ denotes the term resulting from the application of σ (as a substitution) to B. The calculation of instantiations can be performed by a *unification algorithm* for the language. A suitable algorithm has been developed by the first author [Pym90] based on a standard algorithm for simple type theory [Hu75].

A **U**-proof is a pair $\langle\psi, \sigma\rangle$ consisting of a **U**-derivation ψ and an instantiation σ such that $\psi\sigma$ (the application of σ as a substitution to ψ) is an **L**-proof. Not every instantiation that closes the leaves of a given **U**-derivation will yield an **L**-proof when applied. It is sufficient to check that the instantiation can be *well-typed* in the derivation to ensure that the result is an **L**-proof. If Γ is the context and A the type of the universal variable α when introduced into the **U**-derivation, the well-typing condition for α amounts to:

$$\Gamma\sigma \vdash \alpha\sigma{:}\, A\sigma$$

ensuring that side condition (c) of the Πl rule of **L** — the condition omitted from the Πl rule of **U** — is nevertheless satisfied in $\psi\sigma$. The unconstrained choice of term in the Πl rule of **L** is replaced by a highly constrained choice in **U**.[5] This wholesale reduction in the search space is analogous, of course, to that obtained by Robinson in the context of the predicate calculus [Rob65].

The well-typing of an instantiation depends on the structure of the derivation from which it is calculated. However, even if it fails to be well-typed in that derivation it may be well-typed in some permutation of the derivation, since rule applications (or reductions) can sometimes be permuted whilst leaving the endsequent (*i.e.,* root) of the derivation and its leaves unchanged. The degree to which this can be done is summarized in the form of a *Permutation Theorem,* in the sense of Kleene [Kle52] and Curry [Cur52], and underlies the fourth and final system of the paper called **R**.

The rules of **R** are just the rules of **U** (so the derivations are the same) but the condition for instantiations to yield a proof is weakened. An **R**-proof is again a pair consisting of a derivation ψ and an instantiation σ under which ψ is closed, but now we require only that there exist perhaps another (closed) derivation ψ^* in which the instantiation is well-typed; *i.e.,* $\psi^*\sigma$ is an **L**-proof. Of course the crucial computational question is whether the existence of at least one suitable ψ^*,

[5]The inference system that corresponds to the calculation of instantiations by unification does indeed have a *subterm property.* The soundness and completeness result for **U** is a form of *Herbrand Theorem* for the theory. The unification algorithm searches amongst the terms of a "Herbrand universe" defined by each leaf sequent. This aspect is not explored in detail in this paper, see [Pym90].

given ψ and σ, can be determined as the search progresses. We show that this is indeed the case using a *reduction ordering:* a notion which was introduced by Bibel [Bib81] for classical connectives and extended by the second author in [Wal89] to various non-classical connectives.[6] An **R**-proof therefore corresponds to an equivalence class of **U**-proofs of the same endsequent consisting of all permutation variants of the original derivation in which the calculated instantiation is well-typed.

The reader may find it helpful to refer back to the overview given above to identify the motivation for various technicalities below.

2 $\lambda\Pi$-calculus: a theory of dependent function types

The syntax of $\lambda\Pi$-calculus is given by the following grammar:

$$
\begin{array}{ll}
\mathit{Signatures} & \Sigma ::= \langle\rangle \mid \Sigma, c\!:\!K \mid \Sigma, c\!:\!A \\
\mathit{Contexts} & \Gamma ::= \langle\rangle \mid \Gamma, x\!:\!A \\
\mathit{Kinds} & K ::= \mathrm{Type} \mid \Pi x\!:\!A.K \\
\mathit{TypeFamilies} & A ::= c \mid \Pi x\!:\!A.A \mid \lambda x\!:\!A.A \mid AM \\
\mathit{Objects} & M ::= c \mid x \mid \lambda x\!:\!A.M \mid MM
\end{array}
$$

where c ranges over type and object constants. The proof system defined in [HHP87] for deriving assertions of the following forms:

$$
\begin{array}{ll}
\vdash \Sigma \,\mathrm{sig} & \Sigma \text{ is a valid signature} \\
\vdash_\Sigma \Gamma \,\mathrm{context} & \Gamma \text{ is a valid context} \\
\Gamma \vdash_\Sigma K \,\mathrm{kind} & K \text{ is a valid kind} \\
\Gamma \vdash_\Sigma A\!:\!K & A \text{ has kind } K \\
\Gamma \vdash_\Sigma M\!:\!A & M \text{ has type } A
\end{array}
$$

may be found in Appendix 1. We shall refer to this system as **N** to emphasize that it is a system of natural deduction. We stress that **N** is a system of first-order types in the following sense: the Π-type formation rule has the form:

$$
\frac{\Gamma \vdash_\Sigma A\!:\!\mathrm{Type} \quad \Gamma, x\!:\!A \vdash_\Sigma B\!:\!\mathrm{Type}}{\Gamma \vdash_\Sigma \Pi x\!:\!A\,.\,B : \mathrm{Type}}\,;
$$

both A and B must be of kind Type (see also Rule 11 of Appendix 1). There are no variables of kind Type and consequently there are no higher-order types (of kind Type). A summary of the major metatheorems pertaining to **N** and its reduction properties may be found in the Appendix. We note here only that all five relations are decidable.

[6]The condition is equivalent to an enhanced "occurs-check" in the unification algorithm if a suitable notion of *Skolem function* were introduced. The suitable notion is *not* the obvious one that the reader might suppose from experience of classical quantifiers, not least because the logic under investigation here is intuitionistic. In general, the theoretical diversion via Skolemization is unnecessary and can be difficult to justify semantically, even in simple type theory (*cf.* [Mil83]). Our approach follows Herbrand's Theorem (which is finitary) rather than the Skolem-Herbrand-Gödel Theorem (which is not).

3 A metacalculus for N.

DEFINITION 1 (Sequent) A *sequent* is a triple $\langle\Sigma, \Gamma, A\rangle$, written $\Gamma \Rightarrow_\Sigma A$, where Σ is a signature, Γ a context and A a type (family). The intended interpretation of the sequent is the (meta-)assertion:

$$(\exists M) \quad \mathbf{N} \text{ proves } \Gamma \vdash_\Sigma M{:}A.$$

□

We define a semi-logicistic calculus, **L**, for deriving sequents. The system is comprised of two axiom schemata and two operational rules (one left and one right) for the Π-types.

DEFINITION 1 (**L**)

$$Ax1 \quad \Gamma, x{:}A, \Gamma' \Rightarrow_\Sigma A$$

$$Ax2 \quad \Gamma \Rightarrow_{\Sigma, c:A, \Sigma'} A$$

$$\Pi r \quad \frac{\Gamma, x{:}A \Rightarrow_\Sigma B}{\Gamma \Rightarrow_\Sigma \Pi x{:}A.B} \qquad \text{(a) } x \notin \mathrm{Dom}(\Gamma)$$

$$\Pi l \frac{\Gamma, z{:}B[M/x] \Rightarrow_\Sigma C}{\Gamma \Rightarrow_\Sigma C} \qquad \begin{array}{l}\text{(a) } @{:}\, \Pi x{:}A.B \in \Sigma \cup \Gamma \\ \text{(b) } z \notin \mathrm{Dom}(\Gamma) \\ \text{(c) } \mathbf{N} \text{ proves } \Gamma \vdash_\Sigma M{:}A\end{array}$$

Here $B[M/x]$ denotes capture avoiding substitution of M for x, and the conditions $x, z \notin \mathrm{Dom}(\Gamma)$ mean that x and z do not label any declaration in the context Γ. For simplicity and efficiency, we work exclusively with $\beta\eta$-normal forms, and for such terms syntactic identity ($\equiv$) is taken up to α-congruence (change of bound variable). As usual we refer to the variable x of the Πr rule as the *eigenvariable* of the inference. We can ensure that in any derivation eigenvariables occur only in sequents above the inference at which they are introduced. $\Pi x{:}A.B$ is said to be the *principal formula* of the operational rules. A and B are the *side formulae* of the Πr rule, and A and $B[M/x]$ are the side formulae of the Πl rule. **L**-*derivations* are trees of sequents regulated by the operational rules, and **L**-*proofs* are derivations whose leaves are axioms. □

When x is not a free variable of B in $\Pi x{:}A.B$ we have $B[M/x] \equiv B$ and we write $A \to B$ for $\Pi x{:}A.B$: the third side condition on the Πl rule may be weakened to the inhabitation condition:

(c′) $(\exists M) \quad \mathbf{N}$ proves $\Gamma \vdash_\Sigma M{:}A$.

This in turn may be expressed within the system by the sequent $\Gamma \Rightarrow_\Sigma A$ yielding a modified Πl rule which we call $\to l$:

$$\to l \frac{\Gamma \Rightarrow_\Sigma A \qquad \Gamma, z{:}B \Rightarrow_\Sigma C}{\Gamma \Rightarrow_\Sigma C} \qquad \begin{array}{l}\text{(a) } @{:}\, A \to B \in \Sigma \cup \Gamma \\ \text{(b) } z \notin \mathrm{Dom}(\Gamma).\end{array}$$

Consequently, we extend the syntax conservatively to include *non-dependent* function types, $A \to B$, corresponding to $\Pi x{:}A.B$ whenever x is not free in B, and include the $\to l$ rule above and the $\to r$

rule below as derived rules.

$$\to r\frac{\Gamma, x{:}A \Rightarrow_\Sigma B}{\Gamma \Rightarrow_\Sigma A \to B} \quad \text{(a) } x \notin \mathrm{Dom}(\Gamma).$$

DEFINITION 1 (Well-formed sequent) A sequent $\Gamma \Rightarrow_\Sigma A$ is said to be *well-formed* just in case $\Gamma \vdash_\Sigma A$: Type. $\square$

PROPOSITION 2 (HHP87) *The well-formedness problem for sequents is decidable.* $\square$

In practice, derivations are constructed from the root, or *endsequent*, toward the leaves, in the spirit of Kleene [Kle68] and systems of tableaux [Smu68]. In support of this usage we have the following result:

PROPOSITION 3 (PYM90) *For well-formed sequents* $\Gamma \Rightarrow_\Sigma A$*,*

$$\mathbf{L} \text{ proves } \Gamma \Rightarrow_\Sigma A \quad \text{iff} \quad (\exists M) \quad \mathbf{N} \text{ proves } \Gamma \vdash_\Sigma M{:}A.$$

$\square$

We revert to the appropriate fragment of **N** to decide if the endsequent is well-formed. If so, **L** may be utilized to prove inhabitation of A with respect to the context Γ. Moreover an inhabiting term can be extracted from the **L**-proof. Details may be found in [Pym90]. **L** is not fully logicistic since an appeal is still made to **N** for each application of the Πl rule (third side condition).

With the introduction of **L** we have made two conceptual steps. Firstly, we have shifted our attention from a decidable judgement (type assignment) to a semi-decidable judgement (inhabitation) since the former is uninteresting from the point of view of general theorem proving. (Terms code the proofs of their types, hence the decidability of the judgements of **N**.)

The second step concerns proof-search. We moved directly to a sequent system with a limited subformula property. An alternative choice would have been to formulate a natural deduction system for inhabitation assertions which would have given us a Π-elimination rule of the form:

$$\Pi\mathcal{E}\frac{\Gamma \Rightarrow_\Sigma \Pi x : A \,.\, B}{\Gamma \Rightarrow_\Sigma B[M/x]} \quad \text{(a) } \mathbf{N} \text{ proves } \Gamma \vdash_\Sigma M : A$$

similar to the usual natural deduction rule for quantifiers. The fact that the type A in the premiss is not a subformula of the conclusion means that a proof procedure based on such a calculus would have to invent the type. The limited subformula property of **L** restricts the non-determinism to the choice of term M in the Πl rule.

4 A metacalculus for L

We introduce a new syntactic class of *universal* variables denoted by lowercase Greek letters α, β, *etc.*, and extend the syntactic category of objects to include them thus:

$$Objects M ::= c \mid \alpha \mid x \mid \lambda x : A.M \mid MM.$$

Notice that universal variables cannot appear λ-bound. By virtue of this extension, entities of all syntactic classes may now contain universal variables as subterms. When we wish to emphasize that a syntactic entity does not contain universal variables we shall refer to it as being *ground*.

We define a calculus for sequents by dropping the axiom schemata of **L** and modifying the Πl rule as follows:

DEFINITION 3 (**U**-derivation) The rules of **U** consist of the $\to r$, $\to l$ and Πr rules of **L**, together with

$$\Pi l \frac{\Gamma, z{:}B[\alpha/x] \Rightarrow_\Sigma C}{\Gamma \Rightarrow_\Sigma C} \quad \begin{array}{l} \text{(a) } @ : \Pi x{:}A.B \in \Sigma \cup \Gamma \\ \text{(b) } z \notin \mathrm{Dom}(\Gamma). \end{array}$$

U-*derivations* are trees regulated by the above rules such that the sequent at the root of the tree is well-formed. In applications of the Πl rule we call Γ the *typing context* of α and A the *type* of α. (Note that we have not yet defined **U**-*proofs*; there are no axiom schemata.) □

U thus consists of four operational rules. Notice that the Πl rule no longer contains an external appeal to **N** or a choice of term, but is otherwise identical to the Πl rule of **L**.

DEFINITION 3 (Instantiation) An *instantiation* is a mapping from universal variables to objects. The capture-avoiding application of instantiations to all of the constructs of the language is defined in the obvious way. □

The following notion compensates for the absence of axiom schemata in **U**.

DEFINITION 3 (Closure) A sequent $\Gamma \Rightarrow_\Sigma A$ is said to be *closed under* an instantiation σ just in case $B\sigma \equiv A\sigma$ for some declaration $@{:}B \in \Sigma \cup \Gamma$. A **U**-derivation is said to be *closed under* σ just in case all of its leaf sequents are closed under σ. (Again, we work exclusively with $\beta\eta$-normal forms.) □

We are interested in instantiations that are *well-typed* in the following sense.

DEFINITION 3 (Well-typing) An instantiation σ is said to be *well-typed* in a given **U**-derivation just in case for every universal variable α of the derivation, with typing context Γ and type A, we have:

$$\mathbf{N} \text{ proves } \Gamma\sigma \vdash_\Sigma \alpha\sigma{:}A\sigma \qquad \square$$

We are now in a position to define a notion of proof for **U**.

DEFINITION 3 (**U**-proof) A **U**-*proof* is a pair $\langle\psi, \sigma\rangle$, where ψ is a **U**-derivation and σ an instantiation, such that (1) ψ is closed under σ, and (2) σ is well-typed in ψ. □

THEOREM 4.6 *If $\langle\psi, \sigma\rangle$ is a* **U**-*proof, $\psi\sigma$ is an* **L**-*proof.*

PROOF. By induction on the structure of **U**-derivations. The closure condition (1) ensures that the leaves of $\psi\sigma$ are **L**-axioms. The well-typing condition (2) ensures that the image under σ of each instance of a Πl rule (of **U**) in ψ satisfies the side conditions on the Πl rule of **L**. The remaining operational rules are common to the two systems. □

We remark that it is immediate that any **L**-proof arises as a **U**-proof: this is the converse of Theorem 4.6.

We shall postpone discussion of proof-search in **U** until we have introduced our fourth and final refinement. One could stop here, however, using the unification algorithm developed in [Pym90] (based on that of [Hu75]) to calculate instantiations when a putative leaf has been reached, then checking the well-typing condition using **N** (recall that **N** is a decidable system for well-typing). The search space induced by **U** is a proper subspace of that induced by **L** since the choice of term at Πl reductions in the former is constrained by the syntactic content of the leaf sequents (*cf.* [Rob65]). The main reason for considering a further refinement is that **U** distinguishes between derivations that are *intrinsically* identical in a sense made precise below. Consequently, the search space induced by **U** still contains a major source of redundancy.

5 A metacalculus for U

The content of the typing contexts of universal variables in a **U**-derivation depends on the structure of the derivation. For example, the two **U**-derivations below differ in the order in which the Πl and Πr rule have been applied (the figures should be read from endsequent to premisses and we assume that there is some $@ : (\Pi y{:}A.B(y)) \in \Sigma \cup \Gamma$):

$$\dfrac{\dfrac{\Gamma, z{:}B(\alpha), x{:}A \Rightarrow_\Sigma B(x)}{\Gamma, z{:}B(\alpha) \Rightarrow_\Sigma \Pi x{:}A.B(x)}}{\Gamma \Rightarrow_\Sigma \Pi x{:}A.B(x)} \qquad \dfrac{\dfrac{\Gamma, x{:}A, z{:}B(\alpha) \Rightarrow_\Sigma B(x)}{\Gamma, x{:}A \Rightarrow_\Sigma B(x)}}{\Gamma \Rightarrow_\Sigma \Pi x{:}A.B(x)}$$

The principal formula of the Πl reduction in each derivation is the declaration $@ : (\Pi y{:}A.B(y))$ — assumed to be in $\Sigma \cup \Gamma$. We also assume that $x \notin \mathrm{Dom}(\Gamma)$. The instantiation σ that maps α to x closes both derivations. The typing context of α in the first derivation is Γ, while in the second it is $\Gamma, x{:}A$. Since $\alpha\sigma = x$, in order to check the well-typing condition for α we must show $\Gamma \vdash_\Sigma x{:}A$ for the first derivation and $\Gamma, x{:}A \vdash_\Sigma x{:}A$ for the second. Consequently σ is well-typed in the second derivation but not in the first (since $x \notin \mathrm{Dom}(\Gamma)$).

Our final refinement is to introduce a calculus, **R**, in which the existence of the **U**-derivation on the left, together with the closing instantiation, is sufficient to infer the existence of the **U**-proof on the right. That is, we investigate conditions under which rule instances may be *permuted* whilst leaving the endsequent and leaves of the derivation essentially unchanged.

Definition 3 (**R**-derivation) The rules of **R** are exactly the rules of **U**; consequently the derivations of **R** are exactly those of **U**. (The notion of **R**-proof however differs from that of **U**-proof; see below.) $\square$

Let ψ be an **R**-derivation of a given endsequent and let $\mathcal{F}_\psi$ denote the collection of inferences that comprise ψ. We use $\mathcal{F}_\psi(\Xi) \subseteq \mathcal{F}_\psi$ to denote the inferences of a given type Ξ, for Ξ one of the following: $\rightarrow l$, $\rightarrow r$, Πl or Πr. Let σ be an instantiation for ψ.

Definition 3 The following binary relations are defined on $\mathcal{F}_\psi$:

(i) $R <_\psi R'$ iff a side formula of R is the principal formula of R'[7];

(ii) $R \ll_\psi R'$ iff R occurs below R' in ψ;

(iii) $R \sqsubset_\sigma R'$ iff the universal variable or eigenvariable introduced by R is a free variable of $\alpha\sigma$, where α is the universal variable introduced by R'.

□

Notice that $\ll_\psi$ decomposes into sixteen subrelations: $\ll_\psi^{\Xi,\Omega} \subseteq \mathcal{F}_\psi(\Xi) \times \mathcal{F}_\psi(\Omega)$ for Ξ, Ω amongst $\to l$, $\to r$, Πl and Πr. $\ll_\psi$ and its subrelations are called the *skeletal orderings* of the derivation ψ. Notice also that $<_\psi$ is a subrelation of $\ll_\psi$.

DEFINITION 3 (Reduction ordering) The *reduction ordering* $\lhd_{\psi,\sigma}$ induced by an $\mathbf{R}$-derivation ψ and a instantiation σ is defined by:

$$\lhd_{\psi,\sigma} =_{\mathrm{def}} (<_\psi \cup \prec_\psi \cup \sqsubset_\sigma)^+$$

where $+$ indicates transitive closure and the relation $\prec_\psi$ is defined by:

$$\prec_\psi =_{\mathrm{def}} \ll_\psi \ \setminus \ \bigcup_\Xi (\ll_\psi^{\Pi l,\Xi} \cup \ll_\psi^{\Xi,\Pi l});$$

Ξ ranges over the operational rules of $\mathbf{R}$. □

The presence of a relation in $\lhd_{\psi,\sigma}$ indicates that relationship between specific inferences may not be altered by permutation. Consequently, the definition of $\prec_\psi$ fixes the relative positions of all rule applications in ψ *except* Πl rule applications (since they are removed from $\ll_\psi$ to form $\prec_\psi$).

DEFINITION 4(i) (Compatibility) A derivation is said to be *compatible* with an instantiation just in case the reduction ordering induced is irreflexive.

(ii) (Degree) The *degree* of a compatible derivation is the number of pairs of inferences in the derivation whose skeletal order is inconsistent with the reduction ordering. That is, $\langle R, R' \rangle$ for which both $R \ll_\psi R'$ (R is below R' in ψ) and $R' \lhd_{\psi,\sigma} R$. If a derivation is compatible with an instantiation with degree n, we say it is *n-compatible.* □

THEOREM 5.4 (PERMUTATION THEOREM) *If ψ is compatible with σ, then there is a 0-compatible $\mathbf{R}$-derivation ψ^* of the same endsequent. Moreover, if ψ is closed under σ, so is ψ^*. We say that ψ^* is a* permutation *of ψ.*

PROOF. By induction on the degree of ψ. We interchange Πl inferences with other inferences to reduce the degree. One such case was given as an example at the start of this section (Πl over Πr). The others are left to the reader. One must check that the leaves of the permuted derivation contain the same declarations (though in a different order). □

We have as a corollary to the construction performed in the proof of the Permutation Theorem:

[7] More accurately: iff a side formula of R is a "descendent" of the principal formula of R; we distinguish the "occurrences" of a formula in a derivation [Kle68].

Corollary 5 $\lhd_{\psi^*,\sigma} = \lhd_{\psi,\sigma}$.

Proof. The only relationships changed in the permutation are those excluded from $\lhd_{\psi,\sigma}$. $\square$

The following lemma shows that the 0-compatibility of a derivation and instantiation is a necessary condition for the well-typing of the latter in the former.

Lemma 6 *If σ is well-typed in ψ, ψ is 0-compatible with σ.*

Proof. An inconsistency between the skeletal ordering of ψ and the reduction ordering arises from an inconsistency between $\sqsubset_\sigma$ and the skeletal ordering. Since $\sqsubset_\sigma \subseteq \mathcal{F}_\psi(\Xi) \times \mathcal{F}_\psi(\Pi l)$, it must arise from a Πl inference, introducing α say, being nearer the root of the derivation than a Πl or Πr inference that gives rise to a variable, v say, free in $\alpha\sigma$. But then v cannot be declared in the typing context of α since it is introduced above α, and therefore σ is not well-typed, contradicting our hypothesis. Therefore there can be no inconsistencies and ψ is 0-compatible with σ. $\square$

We give a simple example of an **R**-derivation and a closure instantiation which fails to be well-typed in the given derivation, but which is well-typed in a reordering of that derivation.

Let $\Sigma \equiv_{\text{def}} A : \text{Type}, B : A \to \text{Type}, C : \text{Type}, p : A \to \text{Type}, a : A, f : B(a) \to B(a)$, and let $\Gamma \equiv_{\text{def}} x_1 : \Pi x_2 \!:\! A \,.\Pi x_3 \!:\! B(x_2)\,.\, p x_3$.

We search for a proof of the (well-formed) endsequent $\Gamma \Rightarrow_\Sigma \Pi x_4 \!:\! B(a)\,.\, C \to p(fx_4)$. Consider the following **R**-derivation, ψ:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x_6 : \Pi x_3 \!:\! B(\alpha)\,.\, px_3, x_7 : p\beta, x_4 \!:\! B(a), x_5 \!:\! C \Rightarrow_\Sigma p(fx_4)}{\Gamma, x_6 : \Pi x_3 \!:\! B(\alpha)\,.\, px_3, x_7 : p\beta, x_4 \!:\! B(a) \Rightarrow_\Sigma C \to p(fx_4)}\ (\to r)}{\Gamma, x_6 : \Pi x_3 \!:\! B(\alpha)\,.\, px_3, x_7 : p\beta \Rightarrow_\Sigma \Pi x_4 \!:\! B(a)\,.\, C \to p(fx_4)}\ (\Pi r)}{\Gamma, x_6 : \Pi x_3 \!:\! B(\alpha)\,.\, px_3 \Rightarrow_\Sigma \Pi x_4 \!:\! B(a)\,.\, C \to p(fx_4)}\ (\Pi l)}{\Gamma \Rightarrow_\Sigma \Pi x_4 \!:\! B(a)\,.\, C \to p(fx_4)}\ (\Pi l).$$

The leaf sequent is closed by the instantiation $\sigma \equiv \langle\langle a, \alpha\rangle, \langle fx_4, \beta\rangle\rangle$, but fx_4 is not well-typed in $\Gamma, x_6 : \Pi x_3 \!:\! B(a)\,.\, Px_3$. However, if we reorder the rules in the derivation so that the $\Pi\, l$s are used after the $\to r$ and $\Pi\, r$ we obtain the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x_4 \!:\! B(a), x_5 \!:\! C, x_6 : \Pi x_3 \!:\! B(\alpha)\,.\, px_3, x_7 : p\beta \Rightarrow_\Sigma p(fx_4)}{\Gamma, x_4 \!:\! B(a), x_5 \!:\! C, x_6 : \Pi x_3 \!:\! B(\alpha)\,.\, px_3 \Rightarrow_\Sigma p(fx_4)}\ (\Pi l)}{\Gamma, x_4 \!:\! B(a), x_5 \!:\! C \Rightarrow_\Sigma p(fx_4)}\ (\Pi l)}{\Gamma, x_4 \!:\! B(a) \Rightarrow_\Sigma C \to p(fx_4)}\ (\to r)}{\Gamma \Rightarrow_\Sigma \Pi x_4 \!:\! B(a)\,.\, C \to p(fx_4)}\ (\Pi r).$$

This sequent also is closed by the instantiation σ and fx_4 is well-typed in the context $\Gamma, x_4 : B(a), x_5 \!:\! C, x_6 : \Pi x_3 \!:\! B(a)\,.\, px_3$.

From a computational point of view testing for compatibility is a simple matter given a derivation and an instantiation: it is an acyclicity check in a directed graph. Compatibility is not, however, a *sufficient* test for well-typing. The Permutation Theorem gives us the existence of 0-compatible derivations in which we might test for well-typing of the instantiation incrementally (*i.e.,* as it is found) but this involves repeatedly constructing permutations using the constructive proof of the theorem. This is inelegant and computationally expensive.

Another alternative would be to ignore well-typing until a closed, compatible derivation and instantiation have been found, and then utilize the Permutation Theorem once and check well-typing. We reject this option on the grounds that typing constraints reduce the search space of the unification algorithm drastically.

We develop instead a computationally tractable test on a derivation and instantiation that, if passed, guarantees the well-typing of the instantiation in all 0-compatible permutations of the derivation. Our ability to define such a notion is a corollary of the normalization (cut-elimination) result for $\lambda\Pi$-calculus [HHP87] with its attendant subformula property, just as the results obtained in [Bib81] and [Wal89] for other logics rely on metatheorems of this sort.

Henceforth we treat contexts as ordered structures or DAGs rather than sequences since the dependencies between declarations form such an order. Consequently the implicit union, denoted above by a comma, such as in "$\Gamma, x{:}A$", should be understood as an order preserving union of the order (DAG) Γ and the singleton order $x{:}A$. The latter will be higher in the resulting order than the declarations of the free variables in A, and incomparable with the other maximal elements of Γ. This assumption simplifies our discussion.

The following notions are introduced for an **R**-derivation ψ of a well-formed endsequent. We use u, v, w possibly subscripted to denote universal and eigenvariables of ψ. Let $T(v)$ denote the typing expression for the variable v in ψ.

Definition 6 (Intrinsic typing context) The *intrinsic typing context* $I(v)$ for each (eigen- or universal) variable v of ψ is defined inductively on the structure of the endsequent as follows:

$$I(v) =_{\text{def}} \biguplus_{w \in \text{FV}(T(v))} (I(w), w{:}T(w)).$$

$\biguplus$ denotes order-preserving union of orders; $\text{FV}(M)$ denotes the set of free variables of the term M. □

$I(v)$ is well-defined since the endsequent is well-formed. Indeed, we have:

Lemma 7 $I(v) \vdash_\Sigma T(v)$: Type.

Proof. By construction and the well-formedness of the endsequent (see Appendix 1 for the notion of a well-formed context). □

Let ψ be compatible with the instantiation σ. We give an inductive definition of the intrinsic typing context and type of a variable of ψ under a compatible instantiation σ. The induction is on the (well-founded) reduction ordering ($\lhd_{\psi,\sigma}$) over the domain of σ.

Definition 8 ($I_\sigma(v)$ and $T_\sigma(v)$) Base. For all $v \in \text{FV}(\psi)$, define $I_\epsilon(v) = I(v)$ and $T_\epsilon(v) = T(v)$. (ϵ is the empty instantiation.)

Step. Given $v \in \text{Dom}(\sigma)$, we assume that we have defined $I_\sigma(w)$ and $T_\sigma(w)$ for all $w \in \text{Dom}(\sigma)$ such that $w \lhd_{\psi,\sigma} v$ (Inductive Hypothesis). Let $w_1, w_2, \ldots, w_n$ be an enumeration of those variables declared in $I_\epsilon(v)$. By definition of $\lhd_{\psi,\sigma}$ and $I(v)$, we have $w_i \lhd_{\psi,\sigma} v$ for $0 < i < n+1$. Define

$$D_0(v) = I_\epsilon(v) \vdash_\Sigma T_\epsilon(v)\text{:Type}$$
$$D_{k+1}(v) = \text{CUT}\,(\quad I_\sigma(w_k) \vdash_\Sigma w_k\sigma{:}T_\sigma(w_k) \quad , \quad D_k(v) \quad), \qquad 0 \leq k < n.$$

If $D_n(v)$ is the assertion: $\Delta \vdash_\Sigma C$:Type, then define

$$I_\sigma(v) =_{\text{def}} \Delta$$
$$T_\sigma(v) =_{\text{def}} C. \qquad \square$$

The "CUT" operation in the above definition is the admissible rule of transitivity (see Appendix 1). That is, $D_{k+1}(v)$ is defined in

terms of $D_k(v)$ by the following inference figure:

$$\frac{I_\sigma(w_k) \vdash_\Sigma w_k\sigma{:}T_\sigma(w_k) \qquad D_k(v)}{D_{k+1}(v)}\ \text{CUT}.$$

The cut rule is being used to effect substitution of the values (under σ) of universal variables throughout the judgement starting from the "uninstantiated" intrinsic typing context and type. The definition is well-formed since the context of the left premiss of each cut is a subcontext of the right premiss. (This follows from the construction of $I(v)$.) The cut above then serves to eliminate the declaration $w_k{:}T_\sigma(w_k)$ from the context of $D_k(v)$, replacing w_k by $w_k\sigma$ throughout the rest of the assertion.

The enumeration taken is irrelevant since independent cuts commute. Consider

$$\frac{I_\sigma(u_2) \vdash_\Sigma u_2\sigma{:}T_\sigma(u_2) \qquad \dfrac{I_\sigma(u_1) \vdash_\Sigma u_1\sigma{:}T_\sigma(u_1) \qquad D_k(v)}{D_{k+1}(v)}}{D_{k+2}(v)}$$

and

$$\frac{I_\sigma(u_1) \vdash_\Sigma u_1\sigma{:}T_\sigma(u_1) \qquad \dfrac{I_\sigma(u_2) \vdash_\Sigma u_2\sigma{:}T_\sigma(u_2) \qquad D'_k(v)}{D'_{k+1}(v)}}{D'_{k+2}(v)}.$$

In the first derivation $w_k = u_1$ and $w_{k+1} = u_2$. In the second, $w_k = u_2$ and $w_{k+1} = u_1$. If u_1 and u_2 are assumed independent (*i.e.*, unrelated via $\lhd_{\psi,\sigma}$), we have $u_i \notin \text{Dom}(I_\sigma(u_j))$, $i \neq j$. Hence substitution of the value $u_1\sigma$ for u_1 does not interfere with substitution of the value $u_2\sigma$ for u_2, and $D_{k+2} = D'_{k+2}$.

We can now state the desired well-typing condition for σ in ψ.

DEFINITION 8 (Intrinsic well-typing) σ is said to be *intrinsically well-typed* in ψ just in case for all universal variables α of ψ, we have: $\quad I_\sigma(\alpha) \vdash_\Sigma \alpha\sigma{:}T_\sigma(\alpha)$. $\square$

The importance of the definition is summarized by:

PROPOSITION 9 *If σ is intrinsically well-typed in ψ, then it is intrinsically well-typed in all compatible permutations of ψ. In particular it is well-typed in 0-compatible permutations.*

PROOF. Reference to the definition will show that the intrinsic well-typing of σ in ψ does not depend on the Πl structure of ψ. (In fact we deliberately forbade such dependence by our definition of $\lhd_{\psi,\sigma}$.) Hence the conditions are unaffected by permutations allowed by the reduction ordering $\lhd_{\psi,\sigma}$, which is itself unaltered by permutation (Corollary 5). For a 0-compatible permutation ψ^* of ψ, the intrinsic typing context for a variable is a subcontext of the typing context in $\psi^*\sigma$. Since "Thinning" is admissible (Appendix 1), σ is well-typed in ψ^*. $\square$

In a similar vein, we state without proof the following:

PROPOSITION 10 *If σ is well-typed in a 0-compatible derivation ψ, it is intrinsically well-typed in ψ.* $\square$

We can now define a computationally acceptable notion of **R**-proof.

DEFINITION 10 (**R**-proof) An **R**-*proof* is a pair $\langle\psi,\sigma\rangle$ such that (1) ψ is closed under σ; (2) ψ is compatible with σ, and (3) σ is intrinsically well-typed in ψ. $\square$

THEOREM 5.8 *For well-formed sequents $\Gamma \Rightarrow_\Sigma A$,*

$$\mathbf{R} \text{ proves } \Gamma \Rightarrow_\Sigma A \quad \text{iff} \quad \mathbf{U} \text{ proves } \Gamma \Rightarrow_\Sigma A.$$

PROOF. (Only if.) Suppose $\langle\psi,\sigma\rangle$ is an **R**-proof of $\Gamma \Rightarrow_\Sigma A$. The Permutability Theorem gives us a permutation ψ^* of ψ, closed under (hypothesis 1), and compatible with (hypothesis 2), the instantiation σ. Hypothesis (3), via Proposition 9, ensures that σ is well-typed in ψ^*. Hence $\langle\psi^*,\sigma\rangle$ is an **U**-proof.

(If.) Let $\langle\psi,\sigma\rangle$ be an **U**-proof of $\Gamma \Rightarrow_\Sigma A$. By definition ψ is closed under σ and σ is well-typed in ψ. Compatibility follows from Lemma 6, and intrinsic well-typing from Proposition 10. $\square$

6 Some remarks

Instantiations are generated by a unification algorithm acting on putative axiom sequents. They are first checked for compatibility (occurs-check) and then for intrinsic well-typing. The incremental nature of intrinsic typing means that the unification algorithm can use the typing information to constrain its search. New values for previously uninstantiated variables may be used to eliminate those variables from the typing contexts of the remaining ones. No permutations need be calculated.

We have been somewhat cautious in this development and allowed only Πl rules to migrate. As a consequence the basic structure of a derivation is largely fixed. The next step is to remove the ordering constraints induced by the propositional structure of the logic, perhaps using unification here as was done in [Wal89] for first-order intuitionistic logic. The final result would be a *matrix method* in the style of Bibel [Bib81] or Andrews [And81].

Closure instantiations are calculated by unification. In general, the substitutions calculated by the unification algorithm introduce new variables (*i.e.*, variables that are not present in the original context)[8]. However, we require only those substitutions which are well-typed instantiations (under

[8]Indeed, this is true of the basic algorithm for the simply-typed λ-calculus of [Hu75]

some reordering), and so for a a given **R**-derivation ψ we accept (for further analysis) just those substitutions σ which do not introduce new variables. The unification algorithm of [Pym90] is both *sound* and *complete* for the calculation of such instantiations.

7 Non-ground endsequents and logic programming

We have considered ground endsequents and ground instantiations explicitly. The extension of **R** to non-ground endsequents is straightforward. A non-ground sequent together with a *typing constraint* $\mathcal{T}$ for its universal variables is considered to stand for the set of its well-formed ground instances. (The typing

constraint consists of intrinsic typing contexts and types for each universal variable occuring in the endsequent, ensuring that the mutual dependencies do not render any extension of the initial reduction ordering cyclic.) This determines a set $\mathcal{S}(\mathcal{T})$ of ground *answer instantiations*. Any non-ground instantiation calculated from an **R**-proof $\langle\psi, \sigma\rangle$ of the sequent determines a set of ground well-typed *extensions* $\mathcal{S}_\psi(\mathcal{T})$.

We consider non-ground endsequents because they have an interesting logic programming interpretation. A sequent $\Gamma \Rightarrow_\Sigma A$ may be interpreted as a *logic program* in the following sense: Σ determines a language, Γ a list of *program clauses* and A a *query* written in the language Σ. universal variables correspond to *program variables* or *logic variables*; these correspond to the logical variables of the programming language Prolog [CM84]. The whole sequent represents a request to compute a instantiation σ for the universal variables of the sequent such that any ground extension σ' of σ renders the sequent $\Gamma\sigma' \Rightarrow_\Sigma A\sigma'$ **L**-provable. We are exploiting the fact that the underlying lambda calculus of the $\lambda\Pi$-calculus encodes a computation of type A; *i.e.,* a term M for which $\Gamma \vdash_\Sigma M{:}A$. A full discussion of this notion of logic programming, including both operational (as presented here) and model-theoretic semantics, may be found in [Pym90] and in a forthcoming paper by the authors, where we also discuss the application of our techniques to a form of *resolution rule* which generalizes Paulson's higher-order resolution [Pau86] to the $\lambda\Pi$-calculus.

Acknowledgements. The authors are grateful to Anne Salvesen, James Harland, Robert Harper, Furio Honsell, Gordon Plotkin, Randy Pollack and two anonymous referees for helpful suggestions and comments.

References

[And81] Andrews, P.B. Theorem-proving via general matings. *J. Assoc. Comp. Mach.* 28(2):193–214, 1981.

[AHM87] Avron, A., Honsell, F., Mason, I. Using Typed Lambda Calculus to Implement Formal Systems on a Machine. University of Edinburgh, 1987, ECS-LFCS-87-31.

[Bib81] Bibel, W. Computationally Improved Versions of Herbrand's Theorem. In J. Stern, editor, Proc. of the Herbrand Symposium, *Logic Colloquium* '81, pp. 11-28, North-Holland, 1982.

[CM84] Clocksin, W.F., Mellish, C.S. Programming in Prolog, Springer-Verlag, 1984.

[Cur52] Curry, H.B. The permutability of rules in the classical inferential calculus. *J. Symbolic Logic* **17**, pp. 245–248, 1952.

[vDa80] van Daalen, D.T., The language theory of AUTOMATH. PhD thesis, Technical University of Eindhoven, The Netherlands, 1980.

[Gen34] Gentzen, G. Untersuchungen über das logische Schliessen, *Mathematische Zeitschrift* 39 (1934) 176–210, 405-431.

[HHP87] Harper, R., Honsell, F., Plotkin, G. A Framework for Defining Logics. Proc. LICS '87.

[HHP89] Harper, R., Honsell, F., Plotkin, G. A Framework for Defining Logics. Submitted to the J. Assoc. Comp. Mach., 1989.

[Hu75] Huet, G. A Unification Algorithm for Typed λ-calculus. *Theor. Comp. Sci.*, 1975.

[Kle52] Kleene, S.C. Permutability of inferences in Gentzen's calculi *LK* and *LJ*. *Memoirs of the American Mathematical Society* **10**, pp. 1–26, 1952.

[Kle68] Kleene, S.C. Mathematical logic. Wiley and Sons, 1968.

[MR86] Meyer, A. and Reinhold, M. 'Type' is not a type: preliminary report. in Proc. 13th ACM Symp. on the Principles of Programming Languages, 1986.

[Mil83] Miller, D. Proofs in higher-order logic. PhD thesis, Carnegie-Mellon University, Pittsburgh, USA, 1983.

[Ohl88] Ohlbach, H-J. A resolution calculus for modal logics. Proc. 9th Conf. on Automated Deduction, LNCS 310, 1988.

[Pau86] Paulson, L. Natural Deduction Proof as Higher-order Resolution. *J. Logic Programming* **3**, pp. 237–258, 1986.

[Pra65] Prawitz, D. Natural Deduction: A Proof-theoretical Study. Almqvist & Wiksell, Stockholm, 1965.

[Pym90] Pym, D.J. Proofs, Search and Computation in General Logic. PhD thesis. University of Edinburgh, forthcoming.

[Rob65] Robinson, J. A machine-oriented logic based on the resolution principle. *J. Assoc. Comp. Mach.* **12**, pp. 23–41, 1965.

[Sa89] Salvesen, A. In preparation. University of Edinburgh, 1989.

[Smu68] Smullyan, R.M. First-order logic, *Ergebnisse der Mathematik*, Volume **43**, Springer Verlag, 1968.

[Wal89] Wallen, L.A. Automated deduction in non-classical logics, MIT Press, 1989.

Appendix 1

The $\lambda\Pi$-calculus is closely related to the Π-fragment of AUT-PI, a language belonging to the so-called AUTOMATH family. The $\lambda\Pi$-calculus is a language with entities of three levels: *objects*, *types* *and families of types*, and *kinds*. Objects are classified by types, types and families of types by kinds. The kind Type classifies the types; the other kinds classify functions f which yield a type $f(x_1)\ldots(x_n)$ when applied to objects $x_1, \ldots, x_n$ of certain types determined by the kind of f. Any function definable in the system has a type as domain, while its range can either be a type, if it is an object, or a kind, if it is a family of types. The $\lambda\Pi$-calculus is therefore predicative.

The theory we shall deal with is a formal system for deriving assertions of one of the following shapes:

$\vdash \Sigma$ sig	Σ is a signature
$\vdash_\Sigma \Gamma$ context	Γ is a context
$\Gamma \vdash_\Sigma K$ kind	K is a kind
$\Gamma \vdash_\Sigma A{:}K$	A has kind K
$\Gamma \vdash_\Sigma M{:}A$	M has type A

where the syntax is specified by the following grammar:

$$\begin{array}{l}
\Sigma ::= \langle\rangle \mid \Sigma, c : K \mid \Sigma, c : A \\
\Gamma ::= \langle\rangle \mid \Gamma, x : A \\
K ::= \text{Type} \mid \Pi x : A.K \\
A ::= c \mid \Pi x : A.B \mid \lambda x : A.B \mid AM \\
M ::= c \mid x \mid \lambda x : A.M \mid MN
\end{array}$$

We let M and N range over expressions for objects, A and B for types and families of types, K for kinds, x and y over variables, and c over constants. We write AoB for $\Pi x : A.B$ when x does not occur free in B. We refer to the collection of variables declared in a context Γ as $\mathrm{Dom}(\Gamma)$. We assume α-conversion throughout. The inference rules of the $\lambda\Pi$-calculus appear in Table 1.

A term is said to be *well-typed in a signature and context* if it can be shown to either be a kind, have a kind, or have a type in that signature and context. A term is *well-typed* if it is well-typed in some signature and context. The notion of $\beta\eta$-reduction, written $\to_{\beta\eta}$, can be defined both at the level of objects and at the level of types and families of types in the obvious way, for details [HHP89]. $M =_{\beta\eta} N$ iff $M \to^*_{\beta\eta} P$ and $N \to^*_{\beta\eta} P$ for some term P, where * denotes transitive closure. For simplicity we shall write $\to_{\beta\eta}$ for $\to^*_{\beta\eta}$.

Since $\beta\eta$-conversion over $K \cup A \cup M$ is not Church-Rosser, so the order of technical priority in which the basic metatheoretical results are proved is crucial. The theorem below summarizes these results in a convenient order (here α ranges over the basic assertions of the type theory). The reader is referred to [HHP87], [HHP89] and [Sa89] for details of its proof.

Theorem (The Basic Metatheory of the $\lambda\Pi$-calculus)

1. *Thinning is an admissible rule: if* $\Gamma \vdash_\Sigma \alpha$ *and* $\vdash_{\Sigma,\Sigma'} \Gamma, \Gamma'$ context, *then* $\Gamma, \Gamma' \vdash_{\Sigma,\Sigma'} \alpha$.
2. *Transitivity is an admissible rule: if* $\Gamma \vdash_\Sigma M{:}A$ *and* $\Gamma, x : A, \Delta \vdash_\Sigma \alpha$, *then* $\Gamma, \Delta[M/x] \vdash_\Sigma \alpha[M/x]$.
3. *Uniqueness of types and kinds: if* $\Gamma \vdash_\Sigma M{:}A$ *and* $\Gamma \vdash_\Sigma M{:}A'$, *then* $A =_{\beta\eta} A'$, *and similarly for kinds.*

4. *Subject reduction: if* $\Gamma \vdash_\Sigma M : A$ *and* $M \rightarrow^*_{\beta\eta} M'$, *then* $\Gamma \vdash_\Sigma M' : A$, *and similarly for types.*

5. *All well-typed terms are strongly normalizing.*

6. *All well-typed terms are Church-Rosser.*

7. *Each of the five relations defined by the inference system of table 1 is decidable, as is the property of being well-typed.*

8. *Predicativity: if* $\Gamma \vdash_\Sigma M : A$ *then the type-free* λ*-term obtained by erasing all type information from* M *can be typed in the Curry type assignment system.*

9. *Strengthening is an admissible rule: if* $\Gamma, x : A, \Gamma' \vdash_\Sigma \alpha$ *and if* $x \notin \mathrm{FV}(\Gamma') \cup \mathrm{FV}(\alpha)$ *then* $\Gamma, \Gamma' \vdash_\Sigma \alpha$. $\square$

Valid Signature

$$\frac{}{\vdash \langle\rangle \ \ \mathrm{sig}} \tag{127}$$

$$\frac{\vdash \Sigma \ \ \mathrm{sig} \quad \vdash_\Sigma K \ \ \mathrm{kind} \quad c \notin \mathrm{Dom}(\Sigma)}{\vdash \Sigma, c : K \ \mathrm{sig}} \tag{128}$$

$$\frac{\vdash \Sigma \ \ \mathrm{sig} \quad \vdash_\Sigma A\!: \mathrm{Type} \quad c \notin \mathrm{Dom}(\Sigma)}{\vdash \Sigma, c : A \ \mathrm{sig}} \tag{129}$$

Valid Context

$$\frac{\vdash \Sigma \ \ \mathrm{sig}}{\vdash_\Sigma \langle\rangle \ \ \mathrm{context}} \tag{130}$$

$$\frac{\vdash_\Sigma \Gamma \ \ \mathrm{context} \quad \Gamma \vdash_\Sigma A\!: \mathrm{Type} \quad x \notin \mathrm{Dom}(\Gamma)}{\vdash_\Sigma \Gamma, x : A \ \mathrm{context}} \tag{131}$$

Valid Kinds

$$\frac{\vdash_\Sigma \Gamma \ \ \mathrm{context}}{\Gamma \vdash_\Sigma \ \mathrm{Type} \ \ \mathrm{kind}} \tag{132}$$

$$\frac{\Gamma \vdash_\Sigma A\!: \ \mathrm{Type} \quad \Gamma, x : A \vdash_\Sigma K \ \ \mathrm{kind}}{\Gamma \vdash_\Sigma \Pi x : A.K \ \ \mathrm{kind}} \tag{133}$$

Valid Elements of a Kind

$$\frac{\vdash_\Sigma \Gamma \ \ \mathrm{context} \quad c : K \in \Sigma}{\Gamma \vdash_\Sigma c\!: K} \tag{134}$$

$$\frac{\Gamma \vdash_\Sigma A\!: \ \mathrm{Type} \quad \Gamma, x : A \vdash_\Sigma B\!: \ \mathrm{Type}}{\Gamma \vdash_\Sigma \Pi x : A.B\!: \ \mathrm{Type}} \tag{135}$$

$$\frac{\Gamma \vdash_\Sigma A\!: \ \mathrm{Type} \quad \Gamma, x : A \vdash_\Sigma B\!: K}{\Gamma \vdash_\Sigma \lambda x : A.B\!: \Pi x : A.K} \tag{136}$$

$$\frac{\Gamma \vdash_\Sigma B\!: \Pi x : A.K \quad \Gamma \vdash_\Sigma N\!: A}{\Gamma \vdash_\Sigma BN\!: K[N/x]} \tag{137}$$

$$\frac{\Gamma \vdash_\Sigma A\!: K \quad \Gamma \vdash_\Sigma K' \ \ \mathrm{kind} \quad K =_{\beta\eta} K'}{\Gamma \vdash_\Sigma A\!: K'} \tag{138}$$

Valid Elements of a Type

$$\frac{\vdash_\Sigma \Gamma \ \text{ context} \quad c : A \in \Sigma}{\Gamma \vdash_\Sigma c\colon A} \tag{139}$$

$$\frac{\vdash_\Sigma \Gamma \ \text{ context} \quad x : A \in \Gamma}{\Gamma \vdash_\Sigma x\colon A} \tag{140}$$

$$\frac{\Gamma \vdash_\Sigma A\colon \text{ Type} \quad \Gamma, x : A \vdash_\Sigma M\colon B}{\Gamma \vdash_\Sigma \lambda x : A.M\colon \Pi x : A.B} \tag{141}$$

$$\frac{\Gamma \vdash_\Sigma M\colon \Pi x : A.B \quad \Gamma \vdash_\Sigma N\colon A}{\Gamma \vdash_\Sigma MN\colon B[N/x]} \tag{142}$$

$$\frac{\Gamma \vdash_\Sigma M\colon A \quad \Gamma \vdash_\Sigma A'\colon \text{ Type} \quad A =_{\beta\eta} A'}{\Gamma \vdash_\Sigma M\colon A'} \tag{143}$$

Table 1.

The Role of Elimination Inferences in a Structural Framework

Draft (June 14, 1990)

Peter Schroeder-Heister

Universität Tübingen/SNS
Biesingerstr. 10, 7400 Tübingen, F.R.G.
pischroe@dknkurz1.bitnet

Abstract

It is claimed that introduction and elimination inferences operate at different levels. Whereas introduction inferences express logical rules in the genuine sense, elimination inferences are more like structural inferences. This is blurred by certain features of intuitionistic logic, but becomes obvious when systems with weaker structural postulates are considered.

1 The idea of a structural framework

In a sequent-style system of logic we have, according to Gentzen, to distinguish between structural and logical inference schemata. The latter govern the logical content of formulae whereas the former are completely independent of their logical form. An example of a structural inference schema is that of contraction:

$$\frac{X, a, b, Y \vdash c}{X, a, Y \vdash c} \quad ,$$

an example of a logical inference schema that of $\vee$- elimination

$$\frac{X \vdash a \vee b \quad X, a \vdash c \quad X, b \vdash c}{X \vdash c} \quad ,$$

where a and b denote formulae and X and Y denote lists of formulae.[1]

My proposal of "structural frameworks" (see [11]) was to consider logical content to be contained in a database of rules rather than in a collection of logical inference schemata, and to extend the notion of structure so as to cover the way such database rules are handled. So in a structural framework the distinction is not between structural and logical inference schemata but between structural inference schemata (in an extended sense) and logical rules. This gives a more computational reading of logical rules, putting them nearer to clauses in a logic program. It is essential for this approach that premisses of rules may contain implications which belong to the structural side of the framework (called "structural implications") and are thus to be distinguished from logical implications.

[1]In this paper I only consider sequents with a single formula in the succedent. Furthermore, I consider sequent-style systems in the broad sense covering also sequent-style natural deduction. In fact, I basically consider elimination inferences rather than introductions on the left side of the turnstile. Finally, for simplicity I confine myself to propositional logic.

An example of a structural framework for intuitionistic propositional logic would be given by the following syntax, with $\Rightarrow$ as the arrow for structural implications and $\rhd$ as the rule arrow:

$$\begin{array}{lll}
Atom(At) & := Formula\ of\ propositional\ logic & (Letters : a, b, c) \\
Structure(S) & := List\ of\ SI & (Letters : U, X, Y, Z) \\
Structural\ implication(SI) & := At \mid S{\Rightarrow}SI & (Letters : A, B, C) \\
Rule & := S \rhd At & \\
Sequent & := S{\vdash}SI &
\end{array}$$

and the following structural inference schemata:

$$(Refl)\ \frac{}{A{\vdash}A} \qquad (Perm)\ \frac{X, A, B, Y{\vdash}C}{X, B, A, Y{\vdash}C}$$

$$(Contr)\ \frac{X, A, A, Y{\vdash}C}{X, A, Y{\vdash}C} \qquad (Thin)\ \frac{X{\vdash}C}{X, A{\vdash}C}$$

$$({\vdash}{\Rightarrow})\ \frac{X, Y{\vdash}A}{X{\vdash}Y{\Rightarrow}A} \qquad ({\Rightarrow}{\vdash})\ \frac{X{\vdash}Y \qquad Z, A{\vdash}C}{X, Z, (Y{\Rightarrow}A){\vdash}C}$$

$$(\rhd)\ \frac{X{\vdash}Y}{X{\vdash}a}\ Y \rhd a \text{ if Y} \rhd \text{a is in the database}$$

Here $X{\vdash}Y$ is an abbreviation for $\{X{\vdash}A : A \in Y\}$. The database may contain logical rules such as

$$a, b \rhd a\&b \qquad a{\Rightarrow}b \rhd a{\supset}b$$

$$a \rhd a{\vee}b \qquad b \rhd a{\vee}b$$

$$\vdots$$

However, in principle this approach is not confined to logic but the database may contain extra-logical rules.

In contradistinction to previous descriptions ([9, 11]), I now strictly distinguish between structural implications $X{\Rightarrow}A$, which are obtained by iterating $\Rightarrow$ to the left and to the right, and rules $X \rhd A$, which are built up by means of $\rhd$ and have atoms as conclusions. My older concept of higher-level rules, which used $\Rightarrow$ both as a sign for structural implication and as the rule arrow mixed these two things up. It is essential for a computational interpretation of database rules that they may contain structural implications as premisses, but are themselves different from structural implications. The rule arrow $\rhd$ has a unique meaning throughout the various systems we will consider, whereas there will be different structural implications.

This approach gives a natural view of logic programming, if one takes atomic formulae as atoms and considers database rules $A_1, \ldots, A_n \rhd a$ as program clauses. Such a logic programming language would permit structural implications in the bodies of clauses and is insofar an extension of standard approaches (see [5]).

2 Frameworks for systems with weaker structural postulates

Logical systems with weaker structural postulates are systems in which certain structural principles which are standard in the intuitionistic case are not, or not fully available. Examples are BCK-logic, in which contraction is lacking, relevant logics, in which thinning is restricted or modified, linear logic, which has neither contraction nor thinning, and the Lambek calculus which in one of its versions does not even have associativity of the formulae in the antecedent of a sequent. Such systems have received increasing interest, particularly due ot applications of these logics in computer science and theoretical linguistics.

Corresponding structural frameworks can be developed by imposing the restrictions mentioned on the structural inference schemata and by extending the notion of structural implications so as to incorporate these modifications. The logical principles themselves would as before be formulated as database rules. As examples I shall consider (i) associative Lambek-style, (ii) contraction-free, and (iii) relevant structural frameworks.

2.1 A structural framework based on the associative Lambek calculus

In such a framework we have associativity of antecedentia of sequent, i.e., we can write them as lists, but do not have any further structural postulate in the traditional sense apart from $(Refl)$. In the sense of our structural frameworks we do have, of course, further postulates, namely those governing structural implications and the application of rules. Due to the fact that commutativity is lacking, two structural implications are available, which I denote by $\Rightarrow$ and $\Leftarrow$. That means, the syntax of structural implications (SI) is now

$$SI := At \mid S{\Rightarrow}SI \mid SI{\Leftarrow}S.$$

The structural inference schemata governing $\Rightarrow$ and $\Leftarrow$ are then:

$$\frac{X, Y \vdash A}{X \vdash Y{\Rightarrow}A} \qquad\qquad \frac{X\vdash Y \qquad U, A, Z\vdash C}{U, (Y{\Rightarrow}A), X, Z\vdash C}$$

$$\frac{Y, X \vdash A}{X \vdash A{\Leftarrow}Y} \qquad\qquad \frac{X\vdash Y \qquad U, A, Z\vdash C}{U, X, (A{\Leftarrow}Y), Z\vdash C}\ .$$

In addition, we have the schema $(\rhd)$ of rule application, which is formulated as before. Based on these two structural implications, it is possible to include in the database introduction rules for two implications / and \, which were introduced by Lambek and are well-known in categorial grammar:

$$a{\Rightarrow}b \rhd a\backslash b \qquad\qquad b{\Leftarrow}a \rhd a/b\ .$$

It is obvious that the structural framework is essential for which logical constants can be defined in the database. So it is not the database of logical rules which makes the difference betwen logics, but the structural apparatus governing this database.

Apart from a database of logical rules, rules for atomic formulae could be considered and an evaluation procedure for a logic programming system with weaker structural rules could be defined ([10]).

It might be added that the structural framework just given can be extended furthermore, since contraction and thinning are lacking. So everything said in the following two subsections could be applied to the structural framework for the Lambek-system.

2.2 A contraction-free structural framework

Here only the structural schema $(Contr)$ of contraction is dropped as compared to the intuitionistic case, whereas thinning $(Thin)$ and permutation $(Perm)$ are upheld. This gives rise to the distinction between two ways of applying database rules, names according to the schemata

$$(\rhd)\quad \frac{X\vdash A_1\ \ldots\ X\vdash A_n}{X\vdash a}\quad A_1,\ldots,A_n \rhd a$$

(which is what we had before) and

$$(\rhd')\quad \frac{X_1\vdash A_1\ \ldots\ X_n\vdash A_n}{X_1,\ldots,X_n\vdash a}\quad A_1,\ldots,A_n \rhd a\ .$$

Without contraction, these schemata are no longer equivalent.

Defining a rule to be of the form

$$Y_1,,\ldots,,Y_n \rhd a$$

where the Y_i are, as before, lists of structural implications separated by single commas, $(\rhd)$ and $(\rhd')$ could be incorporated as special cases into a more general schema of rule application:

$$(\rhd'')\quad \frac{X_1\vdash Y_1\ \ldots\ X_n\vdash Y_n}{X_1,\ldots,X_n\vdash a}\quad Y_1,,\ldots,,Y_n \rhd a\ .$$

Using this extended syntax of a rule, the database could contain introduction rules for two conjunctions

$$a,b \rhd a\&b \qquad\qquad a,,b \rhd a\wedge b,$$

which by Girard are called additive and multiplicative, respectively.

Again, the structural framework is independent of a specific logical database and can be used as the background to a contraction-free logic programming system. Linear logic (in Girard's terminology: rudimentary linear logic, since modal operators are lacking), in which in addition one does not have thinning, can be dealt with in a similar way.

2.3 A relevant structural framework

Another type of structural framework is obtained if one considers relevance in a system where both extensional and intensional (relevant) implication is available. Such a system would be a double-family system in Belnap's [2] sense based on a binary structuring of premisses of rules and antecedents of sequents. Such systems have especially be investigated by Read and Slaney, based on work by Dunn, Meyer and Giambrone (see references in [6]). What is interesting with such systems as compared to single family systems, where just the schema of thinning is dropped, is that one gets the distribution principle of the relevant system R for free, i.e., it does not have to be added as a separate axiom.

Here we have as the syntax of structural entities, with SI as before for structural implications and S for structures:

$$\begin{array}{ll} S & := SI \mid (S,S) \mid (S;S) \\ SI & := At \mid S{\Rightarrow}SI \mid S{\Rrightarrow}SI \\ \textit{Rule} & := S \rhd At \\ \textit{Sequent} & := S{\vdash}SI\ . \end{array}$$

Various sets of structural inference schemata can be formulated in this framework. The one leading to the standard system of relevant logic R is the following, where $Z[X]$ denotes that X occurs as a substructure in Z and where $X \geq Y$ stands for

$$\frac{Z[X]{\vdash}A}{Z[Y]{\vdash}A}$$

and $X = Y$ for $X \geq Y$ and $Y \geq X$:

$$(X_1,X_2),X_3 = X_1,(X_2,X_3) \qquad\qquad (X_1;X_2);X_3 = X_1;(X_2;X_3)$$

$$X_1,X_2 > X_2,X_1 \qquad\qquad X_1;X_2 > X_2;X_1$$

$$X,X > X \qquad\qquad X;X > X$$

$$X > X,Y$$

$$\frac{X,Y{\vdash}A}{X{\vdash}Y{\Rightarrow}A} \qquad\qquad \frac{X{\vdash}Y \qquad U[A]{\vdash}C}{U[(Y{\Rightarrow}A),X]{\vdash}C}$$

$$\frac{X;Y{\vdash}A}{X{\vdash}Y{\Rrightarrow}A} \qquad\qquad \frac{X{\vdash}Y \qquad U[A]{\vdash}C}{U[(Y{\Rrightarrow}A);X]{\vdash}C}$$

$$\frac{X_1{\vdash}A_1 \qquad X_2{\vdash}A_2}{X_1,X_2{\vdash}a}\ (X_1,X_2) \rhd a \qquad\qquad \frac{X_1{\vdash}A_1 \qquad X_2{\vdash}A_2}{X_1;X_2{\vdash}a}\ (X_1;X_2) \rhd a$$

In the database, one could use the rules

$$a,b \rhd a\&b \qquad\qquad a{\Rightarrow}b \rhd a{\supset}b$$

$$a;b \rhd a{\circ}b \qquad\qquad a{\Rrightarrow}b \rhd a{\to}b$$

to define extensional conjunction (&) and implication ($\supset$) as well as intensional conjunction (fusion, $\circ$) and intensional (relevant) implication ($\to$). In this system the basic difference between extensional (comma) and intensional structuring (semicolon) is that for the latter one thinning is not available.

In a system of relevant logic programming new aspects arise. For example, if the goal $X;Y \vdash c$ is given and the clause $a, b \rhd c$ is in the database, we may compute $\{X \vdash a, Y \vdash b\}$ as the next subgoal, whereas on the basis of the clause $a, b \rhd c$, we may compute the subgoal $\{(X;Y \vdash a), (X;Y \vdash b)\}$. It is obvious that some matching of antecedents of sequents in a goal and bodies of clauses is necessary. Of course, this only makes sense in the context of logic programming with hypothetical queries. For more details see [10, 5].

On the background of the examples of structural frameworks one may develop the idea of a general structural framework that would allow to make all the distinctions mentioned so far. It would then have arbitrarily many families of structural connectives with different structural postulates associated with them, different ways of associating premisses of rules as in ($\rhd''$), perhaps a structural meta-level to handle modal logics (see [3]), and so on. Such a framework would probably not be very easily tractable, apart from the fact that it would definitively have to be open for extensions. The idea of such a general structural framework goes in a different direction than do most logical frameworks, since it comprises a whole world of systems even at the propositional level, still without considering problems of types and quantification logical frameworks normally start with.

3 The problem of elimination rules

So far the examples of logical rules have been only introduction rules. In the structural framework considered in section 1 with unrestricted structural postulates, elimination rules can be added to the database as well. The $\vee$-elimination rule would read

$$a \vee b, (a \Rightarrow c), (b \Rightarrow c) \rhd c \ .$$

Following the pattern of the $\vee$-elimination rule, in [9] I proposed as a general schema for introduction and elimination rules for an n-ary sentential operator s:

$$(S - I \ rules) \quad X_1 \rhd s(a_1, \ldots, a_n) \ \ldots \ X_n \rhd s(a_1, \ldots, a_n)$$

$$(S - E \ rule) \quad s(a_1, \ldots, a_n), (X_1 \Rightarrow c), \ldots, (X_n \Rightarrow c) \rhd c \ .$$

Here the X_i stand for structures (lists of structural implications), in which only formulae built up from $a_1, \ldots, a_n$ and logical operators already defined are allowed to occur. For example, in the case of s being implication, we have $n = 1$ and $X_1 = (a_1 \Rightarrow a_2)$, in the case of disjunction we have $n = 2$ and $X_1 = a_1$, $X_2 = a_2$, in the case of equivalence we have $n = 1$ and $X_1 = ((a_1 \Rightarrow a_2), (a_2 \Rightarrow a_1))$, etc. It can be shown that in all cases the elimination inference given by $(S - E \ rule)$ ist equivalent to the elimination rule one is used to. For example,

$$(\supset E) \quad a \supset b, ((a \Rightarrow b) \Rightarrow c) \rhd c$$

is equivalent to modus ponens

$$a \supset b, a \rhd b \ .$$

According to that view, both introduction and elimination rules are part of the database of logical rules, the elimination rules being generated in a uniform way from the introduction rules.

My original intention when dealing with structural frameworks with weaker postulates was to keep this treatment of logical rules. The program was to consider introduction and elimination rules as parts of the database of logical rules following basically the same uniform pattern in all structural frameworks, whereas the difference in the structural postulates would be responsible for the difference in the logics obtained. In particular, the idea was that the general schema ($S\!-\!E\;rule$) for elimination rules would be the same in all structural frameworks (see [11]).

This program turns out not to be feasible as may be demonstrated by considering the Lambek-style structural framework considered in section 2.1, in comparison to the intuitionistic structural framework considered in section 1. We take the $\vee$-elimination rule as an example. In the intuitionistic framework it has the form

$$(\vee E\; rule_i) \quad a\vee b, (a\Rightarrow c), (b\Rightarrow c) \rhd c$$

and replaces as a database rule the inference schema of $\vee$-elimination

$$(\vee E\; schema_i) \quad \frac{X\vdash a\vee b \qquad X, a\vdash c \qquad X, b\vdash c}{X\vdash c}$$

or equivalently

$$\frac{X\vdash a\vee b \qquad Y, a\vdash c \qquad Y, b\vdash c}{Y, X\vdash c}$$

which one may find in usual sequent-style systems in which there is no conceptual distinction between structural inferences and the database of logical rules.

In a Lambek-style system the situation becomes different. Here the $\vee$-elimination schema has to be formulated as

$$(\vee E\; schema_l) \quad \frac{X\vdash a\vee b \qquad Y[a]\vdash c \qquad Y[b]\vdash c}{Y[X]\vdash c}\;,$$

where the square brackets denote the occurrence of a certain formula within a structure, i.e., at a certain place in a list. Due to the fact that permutation is lacking, $Y[a]\vdash c$ is not necesarily equivalent to $Y, a\vdash c$, as it would be in the intuitionistic case, i.e., ($\vee E\; schema_l$) and ($\vee E\; schema_i$) are not equivalent, but ($\vee E\; schema_i$) is weaker than ($\vee E\; schema_l$). This again means that ($\vee E\; schema_l$) cannot be expressed by the database rules ($\vee E\; rules$). (Similar reasons speak against the rule

$$a\vee b, (a\Leftarrow c), (b\Leftarrow c) \rhd c$$

which might also be considered a candidate in the Lambek system.)

The phenomenon that $Y[a]$ cannot necessarily be replaced by Y, a (or a, Y), which prevents the inference schema of $\vee$-elimination to be formulated as a database rule, is also obtained in the relevant framework. For the relevant framework with structures built up from two binary structural operations denoted by the comma and the semicolon it is obvious that a formula a within a context cannot necessarily be extracted. The same holds also for a contraction-free framework when it is spelled out in more detail than in section 2.2, since then one would have to consider again two ways of structuring the antecedents of sequents, to be denoted by the comma and the double comma. This would in general prevent extraction of formulae from within a context to the right or left side of a context.

There seems to be a way out of this problem, namely by allowing structural implications and not just formulae to be conclusions (heads) of rules. The $\vee$-elimination rule would then have to be formulated as

$$a{\vee}b, (a{\Rightarrow}C), (b{\Rightarrow}C) \rhd C$$

with capital C. If a is in the Lambek system embedded in a context as $Y[a]$, where $Y[a]$ is Z_1, a, Z_2, then the inference schema ($\vee E\ schema_l$) could be translated into a database rule as

$$(\vee E\ rule') \quad a{\vee}b, (a{\Rightarrow}(Z_2{\Rightarrow}c)), (b{\Rightarrow}(Z_2{\Rightarrow}c)) \rhd (Z_2{\Rightarrow}c)\ .$$

This is the approach I followed in [11] (there with respect to relevant logic). However, I do not consider it a feasible way any more since allowing non-atoms as heads of rules would destroy the computational reading of rules, for which atoms as conclusions are essential. Apart from that it is crucial for the translation of ($\vee E\ schema_l$) as ($\vee E\ rule'$) that *Cut* holds at the structural level. This is far from obvious since it depends on the database. To demand *Cut* would heavily restrict the possibility of considering interesting databases outside logic (to be dealt with in a general theory of logic programming). In addition it would be difficult to prove *Cut* for a given extra-logical database. So it would again restrict the computational interpretation of rules.

The situation described reveals in general, quite independently of our notion of a structural framework, a deep asymmetry betwen introduction and elimination inferences. Whereas in ($\vee E\ schema_l$), one has to consider arbitrary embeddings of formulae in structures (denoted by square brackets), this is not the case in introduction schemata. In the Lambek calculus the schema for \-introduction has to be formulated as

$$\frac{X, a{\vdash}b}{X{\vdash}a\backslash b}$$

and *not* as

$$\frac{X[a]{\vdash}b}{X{\vdash}a\backslash b}\ .$$

Under certain conditions, which are quite natural, the latter inference schema would not be conservative over the structural part of the system since it would render permutation derivable. This asymmetry between introduction and elimination rules will be treated in more detail in [7].

This is all blurred in the intuitionistic system with its simple structural assumptions. From the standpoint of a general structural framework, those assumptions are just limiting cases and cannot be taken as universal. Consequences to be drawn for structural frameworks in general therefore also apply to the intuitionistic case.

4 The role of elimination inferences

From the preceding discussions it is clear that elimination inferences cannot belong to the database of rules. In many cases it is impossible to formulate them as rules in an adequate manner. And even if this is possible, as in the intuitionistic case, one should refrain from it in order to keep the conceptual unity of structural frameworks with different structural assumptions.

If elimination inferences cannot be incorporated into the database, how else can they be treated? To formulate them in the ordinary way as inference schemata like ($\vee E\ schema_l$) and add them to the structural inference schemata would destroy the idea of a structural framework, in which the structural part is kept apart from the content (logical or other) which is put into the database.

A way out is indicated by the fact that the elimination inferences follow a general pattern. This general pattern can be formulated as

$$(s - E\ schema) \quad \frac{X \vdash s(a_1, \ldots, a_n) \qquad Y[X_1] \vdash C \quad \ldots \quad Y[X_n] \vdash C}{Y[X] \vdash C}$$

if the introduction rules for s are given by $(s - I\ rules)$. Unlike $(s - E\ rule)$, which we showed to be problematic with restricted structural postulates, $(s - E\ schema)$ would be appropriate for all structural frameworks. Now again, $(s - E\ schema)$ contains too much content to be counted itself as a proper structural rule, because it deals with logical constants s for which introduction rules of a specific form have to be given. However, it can be further generalized to make it independent of the form of database rules, namely by considering it a general elimination schema for arbitrary atoms, not just for logically compound formulae. It then applies to any database of rules with databases consisting of introduction rules for logical constants as a special case. Given a certain database, define

$$\mathbf{D}(a) := \{Z : Z \rhd a\ is\ a\ substitution\ instance\ of\ a\ rule\ in\ the\ database\}.$$

Then the general elimination schema is formulated as

$$(IP) \quad \frac{X \vdash a \qquad \{Y[Z] \vdash C : Z \in \mathbf{D}(a)\}}{Y[X] \vdash C} \ .$$

It is obvious that in the case of logical constants with $(s - I\ rules)$ as introduction rules, $(s - E\ schema)$ is a special case of (IP). However, (IP) is general enough to be counted as a structural postulate as opposed to database rules which govern contents. That does not mean that (IP) has to be an ingredient of any structural framework. As with every structural postulate, one may discuss what happens if it is present and if it is lacking, obtaining various systems in that way. In the case of logic, it represents a way of dealing uniformly with elimination inferences for systems weaker than intuitionistic logic. In the case of other databases, it represents a way of treating arbitrary rules as introduction rules for certain atoms, allowing to invert these rules in a certain way. This is why one may propose to call it an "inversion principle".

It may be noted that (IP) becomes an infinitary schema if $\mathbf{D}(a)$ is infinite. However, natural restrictions can be formulated that make $\mathbf{D}(a)$ finite (see [5]). In the case of logical constants, $\mathbf{D}(a)$ is finite due to the restrictions on introduction rules (such as the one that an X_i in $X_i \rhd s(a_1, \ldots, a_n)$ should contain no formulae except $a_1, \ldots, a_n$).

Computationally, the schema (IP) is not very well tractable since the a in the premisses does not occur in the conclusion. One may therefore propose the schema

$$\frac{\{Y[Z] \vdash C : Z \in \mathbf{D}(a)\}}{Y[a] \vdash C}$$

instead. This is exactly the inference schema which is discussed in the context of logic programming in [5] and [8], upon which the programming language GCLA is based ([1]), and whose first formulation is due to Hallnäs ([4]). In that context it was called the "**D**-rule" or the rule of "local reflection" and was motivated from an entirely different point of view which had nothing to do with structural postulates and restrictions thereof but with considerations concerning inductive definitions and treating databases in that way. The fact that a different approach leads to exactly the same principle confirms its conceptual power.

The view we have arrived at is that of a structural framework which, besides the handling of structural association (of premisses or antecedentia) and of structural implication, may include inversion principles. In the case of logically compound formulae, these inversion principles instantiate to elimination inferences. The (logical or extra-logical) content of formulae is expressed in a database of rules which, in the case of logical composition, are introduction rules in the ordinary sense. When dealing with such an apparatus computationally, a third component will have to be added, namely the level of control. It is still open what this should mean in the case of weak structural systems.

References

[1] Aronsson, M. et al. A survey of GCLA. In: P. Schroeder-Heister (ed.), *Extensions of Logic Programming. Proceedings of the Workshop held in Tübingen, December 1989.* Springer Lecture Notes in Artificial Intelligence, 1990.

[2] Belnap, N. D. Display logic. *Journal of Philosophical Logic*, 11 (1982), 375-417.

[3] Došen, K. Modal logic as metalogic. *SNS-Berichte*, no. 90-5, 1990.

[4] Hallnäs, L. Generalized Horn Clauses. *SICS Research Report*, no. 86003, 1986.

[5] Hallnäs, L. & Schroeder-Heister, P. A proof-theoretic approach to logic programming. *SICS Research Report*, no. 88005, 1988. To appear in revised form in *Journal of Logic and Computation.*

[6] Read, S. *Relevant Logic: A Philosophical Examination of Inference.* Oxford: Basil Blackwell, 1988.

[7] Schroeder-Heister, P. The asymmetry of introduction and elimination rules. In preparation.

[8] Schroeder-Heister, P. Hypothetical reasoning and definitional reflection in logic programming. In: P.Schroeder-Heister (ed.), *Extensions of Logic Programming. Proceedings of the Workshop held in Tübingen, December 1989.* Springer Lecture Notes in Artificial Intelligence, 1990.

[9] Schroeder-Heister, P. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49 (1984), 1284-1300.

[10] Schroeder-Heister, P. Logic programming with weak structural rules. In preparation.

[11] Schroeder-Heister, P. *Structural Frameworks with Higher-Level Rules: Proof-Theoretic Investigations.* Habilitationsschrift. Universität Konstanz, 1987.

Interactive Program Synthesis within ZF set theory

Martin Coen
Computer Laboratory
University of Cambridge

August 7, 1990

Abstract

Program synthesis may be regarded as a theorem proving task. The program to be synthesized is a "logical variable" which is instantiated during the proof of its specification. An attempt to produce a theory for program synthesis is described, by defining a simple programming language of total functions within ZF set theory. The resulting framework allows reasoning about programs using well-founded recursion and induction, and has been used to derive a unification algorithm. The theory has been implemented on machine using Paulson's theorem prover Isabelle.

1 Introduction

A program can be synthesized from a specification S by proving the formula "$?a$ satisfies S", where $?a$ is a logical variable that is instantiated with the program over the course of the proof. The resulting theorem certifies that the synthesized program satisfies the specification. To achieve this a natural deduction style, backwards proof is used. Theorems are like generalized Horn clauses and can be used as rules in a proof. A rule is resolved with a goal by unifying its conclusion with the goal and then replacing the goal by the rule's premises. By repeatedly resolving with appropriate rules, the goal may be reduced to simpler and simpler subgoals until all the subgoals have been solved. If the goal contains logical variables these may be instantiated during unification so that the program is constructed during the proof in the same way that a result is built up during the execution of a Prolog program.

Martin-Löf's type theory[5], MLTT, provides an elegant framework for this style of reasoning. The judgement $?a \in A$ can be read as "$?a$ is a program which satisfies specification A". Proving this formula will instantiate $?a$ with a proof-object that denotes a program. However, there are problems with this. General recursion cannot be handled by the basic theory, though Paulson[9] and Nordström[7] have suggested possible solutions. More significantly, a program derived in MLTT may contain parts which will never be used in computation. These parts are proof-objects which appear as witnesses to the truth of propositions within the specification. Though Nordström and Petersson[8] have suggested using a subset type former to solve this problem, the principle of identifying programs and proof-objects leads to technical difficulties[11].

Other theories exist which admit reasoning about predicates and programs (enriched λ-calculus)

[1]This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

without identifying programs and proof-objects. Dybjer[2] has suggested Aczel's Logical Theory of Constructions[1] as one alternative. Defining the lambda calculus within Zermelo-Fraenkel set theory gives another candidate theory. Section 2 of this report describes the implementation of this extended set theory in Paulson's theorem prover Isabelle[10, 6].

A target programming language $\mathcal{L}$ is considered. $\mathcal{L}$ allows only total functions, so simplifying proofs of termination. It does support general recursion, since using only structural recursion leads to an awkward programming style. The extended set theory can be used as a host logic, $\mathcal{H}$, in which to embed $\mathcal{L}$, so that every program a of $\mathcal{L}$ has a denotation $[\![a]\!] \rho$, in some environment ρ, which is a term in $\mathcal{H}$. Section 3 describes this embedding. Unlike MLTT in which all proof-objects have a computational meaning, only a subset of terms in $\mathcal{H}$ denote programs of $\mathcal{L}$. Therefore synthesis must ensure not only that the constructed term satisfies the given specification but also that the term denotes a program in $\mathcal{L}$. Section 4 describes how specifications are formulated to achieve this. Section 5 describes a calculus of derived rules for synthesizing programs within this framework. As an example Section 6 outlines the synthesis of a simple recursive program.

2 Host Theory

Isabelle allows the syntax and axioms of an object-logic to be specified using its meta-logic. The meta-logic is a subset of intuitionistic higher-order logic and contains abstraction $\%x . \phi$, application $\phi(x)$, implication $\phi \Rightarrow \psi$, universal quantification $\bigwedge x . \phi$ and equality $a \equiv b$. It also allows logical variables, written $?x$, which may be instantiated during proofs. In what follows, outermost meta-level quantifiers may be dropped.

Classical first order logic with equality is defined in the meta-logic with the usual symbols $\forall, \exists, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, =$. On top of this, Zermelo-Fraenkel set theory[12] is defined as follows.

A new binary predicate $\in$ is introduced. The formula $x \in y$ is read "x is a member of y". The predicate $\subseteq$ for subsets is defined by

$$A \subseteq B \;\equiv\; \forall x . x \in A \rightarrow x \in B$$

Using this, an axiom of extensionality is introduced

$$A = B \leftrightarrow A \subseteq B \wedge B \subseteq A$$

However, instead of introducing the usual sum axiom

$$\exists C . \forall x . (x \in C \leftrightarrow \exists B . x \in B \wedge B \in A)$$

a new meta-level function $\bigcup$ is introduced with the axiom

$$\forall x . (x \in \bigcup A \leftrightarrow \exists B . x \in B \wedge B \in A)$$

derivable from the sum axiom and axiom of extensionality, which ensures the necessary uniqueness. In a similar manner axioms are introduced for the meta-level function *Collect* that takes a set and a one-place predicate and constructs a new set using separation, for $\mathcal{P}$ that constructs the power set of a set, for *Upair* that takes two elements and constructs the unordered pair, and for *Replace* that takes a set and a two-place predicate and constructs a new set by replacement. The more suggestive subset notation $\{x \in A, P(x)\}$ will be used instead of *Collect(A,P)*.

This pure set theory is extended with new terms to represent functions. First, ordered pairing is defined by

$$\langle x, y\rangle \;\equiv\; \mathit{Upair}(\mathit{Upair}(x,x), \mathit{Upair}(x,y))$$

Then a meta-level function *Sigma* is defined to construct the disjoint union of a family of sets. Writing *Sigma(A,B)* as $\sum x \in A \,.\, B(x)$ its definition is

$$\begin{aligned}\textstyle\sum x \in A \,.\, B(x) \;\equiv\; \{x \in \mathcal{P}(\mathcal{P}(A \cup \bigcup \mathit{Replace}(A, \%x\ y\,.\,y = B(x)))),\\ \exists a\ b\,.\, a \in A \wedge b \in B(a) \wedge x = \langle a, b\rangle\}\end{aligned}$$

The meta-level function $\prod$ is defined to construct the cartesian product of a family of sets.

$$\begin{aligned}\textstyle\prod x \in A \,.\, B(x) \equiv \{R \in \mathcal{P}(\sum x \in A \,.\, B(x)),\\ (\forall x\ y\ z\,.\,\langle x,y\rangle \in R \wedge \langle x,z\rangle \in R \supset y = z) \;\wedge\; (\forall x\,.\, x \in A \supset \exists y\,.\,\langle x,y\rangle \in R)\}\end{aligned}$$

Defining lambda abstraction, λ, by

$$\lambda x \in A \,.\, b(x) \equiv \mathit{Replace}(A, \%x\ y\,.\,y = \langle x, b(x)\rangle)$$

and application, ‘, by

$$f`a \equiv \bigcup\{b \in \bigcup\bigcup f, \langle a, b\rangle \in f\}$$

allows the following theorems to be derived

$$\frac{b(x) \in B(x) \qquad [x \in A]}{(\lambda x \in A \,.\, b(x)) \;\in\; (\prod x \in A \,.\, B(x))} \qquad (\Pi\_intr)$$

$$\frac{\begin{array}{l} a \in A \\ f \in \prod x \in A \,.\, B(x)\end{array}}{f`a \in B(a)} \qquad (\Pi\_elim)$$

as well as rules for β and η-conversion. Note that theorems are written as natural deduction rules. Premises appear one above another, and a hypothetical premise is followed by a list of its assumptions in square brackets.

Continuing in this manner, a type theory can be developed within the set theory. New sets are defined to represent the types of booleans *bool* and natural numbers *nat*. New meta-level functions are defined to construct products $A * B$, functions $A \to B$, disjoint unions $A + B$, and lists *List(A)*. The sets that can be generated from *bool* and *nat* using these meta-level functions are called *simple types*. The constructors for the simple types are *true*, *false*, *zero*, *succ*, λ, $\langle \_, \_\rangle$, *inl*, *inr*, *nil*, *cons*; and the eliminators for them are *cond*, *natcase*, ‘, *paircase*, *pluscase*, *listcase*.

From these definitions, introduction and elimination rules can be derived for each simple type. As an example, those for *List(A)* are

$$nil \in List(A) \qquad (nil\_intr)$$

$$\frac{\begin{array}{l} h \in A \\ t \in List(A) \end{array}}{cons(h,t) \in List(A)} \qquad (cons\_intr)$$

$$\frac{\begin{array}{l} l \in List(A) \\ b \in B(nil) \\ c(h,t) \in B(cons(h,t)) \qquad [h \in A;\ t \in List(A)] \end{array}}{listcase(l,b,c) \in B(l)} \qquad (listcase\_intr)$$

Note that the rules for eliminators, like *listcase* above, are just instances of case analysis over their type. Unlike the corresponding rules of MLTT they do not provide structural recursion.

Instead a meta-level function for general recursion is defined

$$wrec(\prec,\ \%x\ g\ .\ h(x,g),\ a)$$

where:

$\prec$	is a well-founded relation
$\%x\ g\ .\ h(x,g)$	is a function representing the body of the recursion where x represents the argument and g a recursive call
a	is the argument of the function

Its definition is such that the following are theorems:

$$\frac{\begin{array}{l} wfrel(A,\prec) \\ a \in A \\ h(x,g) \in B(x) \qquad [x \in A;\ g\ \in\ \prod y \in A \downarrow_{\prec x} .B(y)] \end{array}}{wrec(\prec,h,a) \in B(a)} \qquad (wrec\_intr)$$

$$\frac{\begin{array}{l} wfrel(A,\prec) \\ a \in A \\ h(x,g) \in B(x) \qquad [x \in A;\ g\ \in\ \prod y \in A \downarrow_{\prec x} .B(y)] \end{array}}{wrec(\prec,h,a) = h(a, \lambda x \in A \downarrow_{\prec a} .wrec(\prec,h,x))} \qquad (wrec\_conv)$$

where $A \downarrow_{\prec a} \equiv \{x \in A,\ x \prec a\}$ and $wfrel(A,\prec)$ iff $\prec$ is a well-founded relation over the set A. The theorem *wrec_intr* is an instance of well-founded induction over the relation $\prec$.

Two more sets are defined. $PTYPE$ is the set of all simple types. $ETYPE$ is the set of all simple types that do not involve functions, that is those for which equality is decidable. Both can be proved to exist in this set theory.

In the following, $\mathcal{H}$ is used to refer to the theory consisting of first order logic with the axioms of Zermelo-Fraenkel set theory and extended with all the definitions above.

3 Target Language

$\mathcal{L}$ is defined using Isabelle's meta-logic, which allows operators to be meta-level functions. A semantic function, $[\![\ \_\ ]\!]\ \_$, maps terms in $\mathcal{L}$ of type *Prog*, in an environment of type *Env*, to terms in $\mathcal{H}$ of type *Term*. The target language, $\mathcal{L}$, permits only total functions, but provides general recursion. Achieving this combination requires annotations on the operators for λ-abstraction and recursion in $\mathcal{L}$ that are sets from $\mathcal{H}$.

A program variable $\mathbf{v}_i$, labelled with a natural number i, is given a denotation using an environment in the standard way (see Appendix A).

$$[\![\ \mathbf{v}_i\ ]\!]\ \rho \ \equiv\ \rho(\mathbf{v}_i)$$

The operator for λ-abstraction has type

$$\mathbf{lam}_A\ :\ (Prog \to Prog) \to Prog$$

It is annotated with a set A, from the host logic, which is not used in computation but allows the denotation to be a total function. Its role is akin to that of a type in a statically typed programming language.

$$[\![\ \mathbf{lam}_A x\ .\ b(x)\ ]\!]\ \rho \ \equiv\ \lambda x \in A\ .\ [\![\ b(\mathbf{v}_i)\ ]\!]\ \rho[x/\mathbf{v}_i] \qquad\qquad (A \in PTYPE)$$

Note that for any rule in which a new program variable is introduced such as $\mathbf{v}_i$ above, $\mathbf{v}_i$ is assumed to be a new variable that does not appear in ρ. This condition is not explicitly mentioned again.

The operator for recursion is annotated with a set $\prec$, representing a well-founded relation. This relation is necessary for the denotation to be a total function (see the theorems *wrec_intr* and *wrec_conv* above). An interpreter for this language need not examine the relation $\prec$ when evaluating the term $\mathbf{wrec}_{\prec}(h, a)$, since a synthesized program is guaranteed to terminate and its value only depends on the body of the recursion, h, and the initial argument, a.

$$[\![\ \mathbf{wrec}_{\prec}(h,a)\ ]\!]\ \rho \ \equiv\ wrec(\ \prec,\ \%x\ g\ .\ [\![\ h(\mathbf{v}_i, \mathbf{v}_j)\ ]\!]\ \rho[x/\mathbf{v}_i][g/\mathbf{v}_j],\ [\![\ a\ ]\!]\ \rho)$$

Though the language is typeless it contains constructors and eliminators to support the simple types. Constructors are defined in the standard denotational manner, for example

$$[\![\ \mathbf{succ}(n)\ ]\!]\ \rho \ \equiv\ succ(\ [\![\ n\ ]\!]\ \rho\ )$$

Eliminators are used rather than projection operators, such as *first* and *second*, as they make reasoning easier and, arguably, lead to a better programming style. The eliminators resemble ML pattern matching that is disjoint and exhaustive. They take meta-level functions as arguments and so their denotations involve extending some environments, for instance

$$[\![\ \mathbf{natcase}(n,b,c)\ ]\!]\ \rho \ \equiv\ natcase(\ [\![\ n\ ]\!]\ \rho,\ [\![\ b\ ]\!]\ \rho,\ \%x\ .\ [\![\ c(\mathbf{v}_i)\ ]\!]\ \rho[x/\mathbf{v}_i]\ )$$

There is also a built-in function for equality over types that are members of *ETYPE*. The complete semantics is given as Appendix A.

4 Specifications

The specification "a satisfies S" is represented in $\mathcal{H}$ by the formula

$$[\![\, a \,]\!]\, \rho_0 \in S$$

where ρ_0 is the empty environment.

Specifications may be loose, allowing many possible implementations. Representing them as sets is a natural way of capturing this ambiguity. A function in $\mathcal{L}$ is specified with a set $\prod x \in A . B(x)$. An input condition of the specification is expressed by restricting A to a subset of a simple type, and an output condition by restricting $B(x)$. As an illustration, a specification of a curried function that returns the integer quotient of one natural number divided by a second non-zero natural number is

$$[\![\, ?a \,]\!]\, \rho_0 \;\in\; \prod x \in nat \,.\, \prod y \in \{n \in nat,\; n \neq 0\} \,. \\ \{z \in nat,\; \exists r \,.\, r \in nat \wedge r < y \wedge x = z \times y + r\}$$

5 A Calculus for Synthesis

A set of theorems can be derived in $\mathcal{H}$ that provides a calculus of rules for synthesizing programs. A rule to introduce lambda abstraction into a program could be

$$\frac{\begin{array}{l} A \in PTYPE \\ [\![\, b(\mathbf{v}_i) \,]\!]\, \rho[x/\mathbf{v}_i] \;\in\; B(x) \qquad [x \in A] \end{array}}{[\![\, \mathbf{lam}_A x \,.\, b(x) \,]\!]\, \rho \;\in\; \prod x \in A \,.\, B(x)}$$

Introduction rules, similar to the one above, can be derived for all the operators of $\mathcal{L}$. However, the resulting calculus is awkward to use as it requires explicit reasoning about the environment. If a new predicate $\vdash$ is introduced such that $a \vdash S$ iff a is the denotation of a program and $a \in S$, then a much simpler calculus can be derived for the synthesis of program denotations. The rule for the denotation of a program lambda abstraction is

$$\frac{\begin{array}{l} A \in PTYPE \\ b(x) \vdash B(x) \qquad [x \in A] \end{array}}{\lambda x \in A \,.\, b(x) \;\vdash\; \prod x \in A \,.\, B(x)} \qquad (lam\_rule)$$

The synthesis rule for program variables is simply

$$\frac{a \in A}{[\![\, \mathbf{v}_i \,]\!]\, \rho[a/\mathbf{v}_i] \in \{x \in A, x = a\}}$$

However, the rule for the denotation, using the new predicate, requires a side condition to ensure that the term being considered is actually part of the current environment.

$$\frac{a \in A}{a \vdash A} \quad \textit{(a denotes a program variable)} \qquad (var\_rule)$$

For a functional program, variables appear in the environment under the same conditions as variables in natural deduction appear in scope. So the side condition is satisfied iff a is a universally quantified meta-variable. To implement the rule in Isabelle, a tactic (*var_tac*) must be used that checks this side condition before applying the rule.

The other rules do not present any problems. The rule for *wrec* introduces an induction hypothesis that can be used in synthesizing the body of the recursion.

$$\frac{\begin{array}{ll} a \vdash A & \\ h(x,g) \vdash B(x) & [x \in A;\ g \vdash \prod y \in A \downarrow_{\prec x} .B(y)] \\ \mathit{wfrel}(A, \prec) & \end{array}}{\mathit{wrec}(\prec, h, a) \vdash B(a)} \qquad (\mathit{wrec\_rule})$$

The rule for an eliminator breaks the goal into the appropriate cases for that simple type, for example

$$\frac{\begin{array}{ll} l \vdash \mathit{List}(A) & \\ b \vdash B(\mathit{nil}) & \\ c(h,t) \vdash B(\mathit{cons}(h,t)) & [h \in A;\ t \in \mathit{List}(A)] \end{array}}{\mathit{listcase}(l, b, c) \vdash B(l)} \qquad (\mathit{listcase\_rule})$$

The rule for a constructor is no more than simple type checking

$$\frac{\begin{array}{l} h \vdash A \\ t \vdash \mathit{List}(A) \end{array}}{\mathit{cons}(h,t) \vdash \mathit{List}(A)} \qquad (\mathit{cons\_rule})$$

This calculus of rules provides the only way to reduce goals of the form $a \vdash A$. There is exactly one rule for each operator that a represents, so to solve a goal in which a is instantiated there is only one possible sequence of rule applications. The rules for the eliminators and the recursion operator are applicable for all specifications A and correspond to case analysis and induction respectively. However, the rules for constructors are applicable only when A is a simple type. As specifications combine a simple type and a proposition, a further rule, derivable from subset introduction, is needed to separate these two parts.

$$\frac{\begin{array}{l} P(a) \\ a \vdash A \end{array}}{a \vdash \{x \in A, P(x)\}} \qquad (\mathit{spec\_rule})$$

For a specification $\{x \in A, P(x)\}$, A is a simple type and so the second premise can be solved using the existing rules. By the very nature of a synthesis proof a will be uninstantiated and so there is a choice in applying a synthesis rule. However, if *spec_rule* is applied and the first premise solved then a will be instantiated (probably) in the second premise. A goal $a \vdash A$ in which a is instantiated and A is a simple type is decidable. An automatic tactic can be used that will solve this goal as a becomes instantiated during the proof of $P(a)$. This will ensure that only type correct instantiations are made to a during the proof of $P(a)$.

6 Deriving a Program

To illustrate the use of this calculus, an example synthesis is outlined below. In this proof, subgoals that could be solved automatically as described above will be handled explicitly. A simple recursive function to append two lists is considered. Although this function only requires structural recursion, it should be apparent from the proof that this method is suitable for more general recursion as well. A specification of an append function is

$$A \in PTYPE \quad \supset \; ?app \vdash \prod k \in List(A) \, . \prod l \in List(A) \, . \{m \in List(A), Append(k,l,m)\}$$

where $Append(k,l,m)$ is a predicate which is valid iff m is the list obtained by appending l to the end of k. To avoid cluttering the proof with extra detail the following properties, which correspond to a Prolog implementation, will be assumed for this predicate.

$$Append(nil,l,l) \tag{App_nil}$$

$$Append(t,l,x) \supset Append(cons(h,t),l,cons(h,x)) \tag{App_cons}$$

The current proof state is presented together with the state of program instantiation in the following form

State O

$$\left[\begin{array}{ll} \text{Program:} & ?app \\ \text{Subgoals:} & \\ & \\ 1. & ?app \vdash \prod k \in List(A) \, . \prod l \in List(A) \, . \{m \in List(A), Append(k,l,m)\} \end{array}\right.$$

The only way to proceed when the specification is a $\prod$-set is by using *lam_rule*.

State I

$$\left[\begin{array}{ll} \text{Program:} & \lambda a \in List(A).?b(a) \\ \text{Subgoals:} & \\ & \\ 1. & List(A) \in PTYPE \\ & \\ 2. & \dfrac{k \in List(A)}{?b(k) \vdash \prod l \in List(A) \, . \{m \in List(A), Append(k,l,m)\}} \end{array}\right.$$

From the definition of $PTYPE$, the rule $A \in PTYPE \Rightarrow List(A) \in PTYPE$ is derivable. Applying this followed by the premise $A \in PTYPE$ solves goal (1). For subgoal (2), *lam_rule* is again the only alternative, giving

State II

$$\left[\begin{array}{ll}
\text{Program:} & \lambda a \in \mathit{List}(A) \,.\, \lambda b \in \mathit{List}(A) . ?c(a,b) \\
\text{Subgoals:} & \\
& \\
1. & \mathit{List}(A) \in \mathit{PTYPE} \\
& \\
2. & \dfrac{\begin{array}{l} k \in \mathit{List}(A) \\ l \in \mathit{List}(A) \end{array}}{?c(k,l) \ \vdash \ \{m \in \mathit{List}(A), \mathit{Append}(k,l,m)\}}
\end{array}\right.$$

Subgoal (1) is solved as before. Now *wrec_rule* is applied to (2) to introduce an induction hypothesis for deriving the function's body. The rule for *wrec* allows induction to be considered over a tuple of all the function's arguments. A tactic combining the repeated use of *lam_rule* with *wrec_rule* provides a standard beginning to the derivation of a general recursive function. To simplify this example, induction will be considered over just the first argument, k.

State III

$$\left[\begin{array}{ll}
\text{Program:} & \lambda a \in \mathit{List}(A) \,.\, \lambda b \in \mathit{List}(A) \,.\, \mathit{wrec}(?{\prec}, ?h(a,b), a) \\
\text{Subgoals:} & \\
& \\
1. & k \vdash \mathit{List}(A) \\
& \\
2. & \dfrac{\begin{array}{l} x \in \mathit{List}(A) \\ g \ \in \ \prod y \in A \downarrow_{?\prec x} . \{m \in \mathit{List}(A), \mathit{Append}(y,l,m)\} \end{array}}{?h(k,l,x,g) \ \vdash \ \{m \in \mathit{List}(A), \mathit{Append}(x,l,m)\}} \\
& \\
3. & \mathit{wfrel}(\mathit{List}(A), ?{\prec})
\end{array}\right.$$

Note that the well-founded relation, $?{\prec}$, is uninstantiated. Goal (2) can be solved using the induction hypothesis and a relation chosen afterwards to fit the requirements. This approach, used successfully by Manna and Waldinger in their deductive tableau system [4], provides a very flexible mechanism for handling recursion.

var_tac succeeds in solving subgoal (1) as k is a meta-variable and $k \in \mathit{List}(A)$ is an assumption. *listcase_rule* splits a subgoal into two cases, for *nil* and *cons*. If the rule is applied to subgoal (2) for variable x, the first premise $x \vdash \mathit{List}(A)$ can be solved by *var_tac*, leaving

State IV

$$\left[\begin{array}{ll}
\text{Program:} & \lambda a \in \mathit{List}(A)\,.\,\lambda b \in \mathit{List}(A)\,.\,\mathit{wrec}(?\prec, \\
& \qquad \%x\ g.\mathit{listcase}(x, ?t(a,b,x,g), ?u(a,b,x,g)), a) \\
\text{Subgoals:} & \\
1. & \dfrac{g\ \in\ \prod y \in \mathit{List}(A) \downarrow_{?\preceq nil}.\{m \in \mathit{List}(A), \mathit{Append}(y,l,m)\}}{?t(k,l,x,g)\ \vdash\ \{m \in \mathit{List}(A), \mathit{Append}(nil,l,m)\}} \\
2. & \begin{array}{l} h \in A \\ t \in \mathit{List}(A) \\ \hline \end{array} \\
 & \dfrac{g\ \in\ \prod y \in \mathit{List}(A) \downarrow_{?\preceq cons(h,t)}.\{m \in \mathit{List}(A), \mathit{Append}(y,l,m)\}}{?u(k,l,x,g,h,t)\ \vdash\ \{m \in \mathit{List}(A), \mathit{Append}(cons(h,t),l,m)\}} \\
3. & \mathit{wfrel}(\mathit{List}(A), ?\prec)
\end{array}\right.$$

Applying *spec_rule* to (1) produces the subgoals

$$\mathit{Append}(nil, l, ?t(k,l,x,g))$$
$$?t(k,l,x,g) \vdash \mathit{List}(A)$$

Applying *App_nil* solves the first of these to leave $l \vdash \mathit{List}(A)$ which may be solved using *var_tac*. Applying *spec_rule* to (2) and then applying *App_cons* to the first premise gives

State V

$$\left[\begin{array}{ll}
\text{Program:} & \lambda a \in \mathit{List}(A)\,.\,\lambda b \in \mathit{List}(A)\,.\,\mathit{wrec}(?\prec, \\
& \qquad \%xg.\mathit{listcase}(x, b, \%ht\,cons(h, ?u(k,l,x,g,h,t))), a) \\
\text{Subgoals:} & \\
 & h \in A \\
 & t \in \mathit{List}(A) \\
1. & \dfrac{g\ \in\ \prod y \in \mathit{List}(A) \downarrow_{?\preceq cons(h,t)}.\{m \in \mathit{List}(A), \mathit{Append}(y,l,m)\}}{\mathit{Append}(t, l, ?u(k,l,x,g,h,t))} \\
2. & cons(h, ?u(k,l,x,g,h,t))\ \vdash\ \mathit{List}(A) \\
3. & \mathit{wfrel}(\mathit{List}(A), ?\prec)
\end{array}\right.$$

The following rule can be derived from Π-*elim*

$$\frac{\begin{array}{l} g\ \in\ \prod y \in A \downarrow_{\prec x}.\{z \in B, P(y,z)\} \\ Q \qquad [g`a \vdash B;\ P(a, g`a)] \\ a \vdash A \\ a \prec x \end{array}}{Q}$$

Applying this to (1) and solving the first premise by assumption gives

State VI

Program: $\lambda a \in List(A) . \lambda b \in List(A) . wrec(?\prec,$
$\%xg.listcase(x, b, \%ht cons(h, ?u(k, l, x, g, h, t))), a)$

Subgoals:

1. $$\frac{\begin{array}{l} g`?a \vdash List(A) \\ Append(?a, l, g`?a) \end{array}}{Append(t, l, ?u(k, l, x, g, h, t))}$$

2. $?a \vdash List(A)$

3. $?a \;?\!\prec cons(h, t)$

4. $cons(h, ?u(k, l, x, g, h, t)) \;\vdash\; List(A)$

5. $wfrel(List(A), ?\prec)$

Goal (1) is solved by assumption, instantiating $?u(k, l, x, g, h, t) \leftarrow g`t$. Goal (2), now $t \vdash List(A)$, is solved by *var_tac*, leaving

State VII

Program: $\lambda a \in List(A) . \lambda b \in List(A) . wrec(?\prec,$
$\%xg.listcase(x, b, \%h\ t . cons(h, g`t)), a)$

Subgoals:

1. $t \;?\!\prec\; cons(h, t)$

2. $cons(h, g`t) \vdash List(A)$

3. $wfrel(List(A), ?\prec)$

Using *cons_rule*, *var_tac* and the premise $g`t \;\vdash\; List(A)$, goal (2) can be solved. The remaining two goals constrain the choice of relation for the recursion. They can be solved using the relation *TAIL(A)*, where a *TAIL(A)* b iff the list a is the tail of the list b. The resulting program denotation is

$$\lambda a \in List(A) . \lambda b \in List(A) . wrec(TAIL(A), \%x\ g . \mathrm{listcase}(x, b, \%h\ t . \mathrm{cons}(h, g`t), a))$$

For this example, the well-founded relation used is precisely the relation implicit in structural induction over lists. The advantage of using *wrec* is that the treatment of recursive calls, as illustrated above, remains the same when the choice of relation becomes more elaborate.

7 Conclusion

The calculus described above has been implemented using the theorem prover Isabelle, though at present some derivable theorems have been assumed. Using this implementation, several simple functions over booleans and lists have been synthesized, as well as programs for division of natural numbers, sorting and first-order unification. The examples are not proper syntheses as many convenient lemmas have been assumed rather than deriving all of the necessary background theory. However, they do show some nice features of the calculus.

Since $\mathcal{L}$ supports general recursion, algorithms in the target programming language closely resemble those in conventional functional languages, such as (functional) ML. Synthesis proofs follow this same structure, using well-founded induction (for general recursion) followed by case analysis (for pattern matching). Even a function for first-order unification, which contains nested recursive calls, can be synthesized with a single induction step. The well-founded relation needed in this case is a lexicographic ordering of a subtree relation and a subset relation over variables. This relation would be an obvious choice in proving termination of the unification function.

This calculus has the flavour of a simple type theory. It does not identify propositions and types, so avoiding redundant proof-objects in programs. Subsets allow the use of propositions to tailor specifications more finely than pure simple typing would allow. This use of subsets in a classical extensional theory is closely related to Salvesen's[11] observation that a subset type former can be added to an extensional type theory with additional rules for the law of the excluded middle. The $\sum$-set provides a natural model for a **let** construct in $\mathcal{L}$. Using this to structure syntheses into modules remains to be investigated.

A Definition of Target language

A.1 Syntax

There are four types of term in the following syntax:

Nat	natural numbers
$Term$	terms of the host theory
$Prog$	terms of the target language
Id	identifiers of the target language ($Id \subset Prog$)

Operators of the target language are higher-order functions of the following types:

$$
\begin{aligned}
\mathbf{true}&:Prog\\
\mathbf{false}&:Prog\\
\mathbf{zero}&:Prog\\
\mathbf{succ}&:Prog \to Prog\\
\mathbf{lam}&:Term * (Prog \to Prog) \to Prog\\
\mathbf{pair}&:Prog * Prog \to Prog\\
\mathbf{inl}&:Prog \to Prog\\
\mathbf{inr}&:Prog \to Prog\\
\mathbf{nil}&:Prog\\
\mathbf{cons}&:Prog * Prog \to Prog\\
\mathbf{cond}&:Prog * Prog * Prog \to Prog\\
\mathbf{natcase}&:Prog * Prog * (Prog \to Prog) \to Prog\\
\mathbf{app}&:Prog \to Prog\\
\mathbf{paircase}&:Prog * (Prog * Prog \to Prog) \to Prog\\
\mathbf{pluscase}&:Prog * (Prog \to Prog) * (Prog \to Prog) \to Prog\\
\mathbf{listcase}&:Prog * Prog * (Prog * Prog \to Prog) \to Prog\\
\mathbf{wrec}&:Term * (Prog * Prog \to Prog) * Prog \to Prog\\
\mathbf{eq}&:Prog\\
\mathbf{v}&:Nat \to Id
\end{aligned}
$$

A.2 Semantics

Environments are defined in the usual way as a mapping from Id to $Term$,

$$\rho \ : \ Env = Id \to Term$$

New environments are formed from old using square brackets, which have the following meaning

$$\rho[x/\mathbf{v}_i](\mathbf{v}_j) = \begin{cases} x & \text{if } i = j \\ \rho(\mathbf{v}_j) & \text{otherwise} \end{cases}$$

The semantic function is of type

$$[\![\ \_\ ]\!]\ \_ : Prog \to Env \to Term$$

Two sets are predefined for use in the semantics that follow:

$PTYPE$	set of all simple types
$ETYPE$	set of all types for which the built in function **eq** is defined

The meanings of the operators are:

$$[\![\ \mathbf{true}\ ]\!]\ \rho \ \equiv\ \mathit{true}$$

$$[\![\ \mathbf{false}\ ]\!]\ \rho \ \equiv\ \mathit{false}$$

$$[\![\ \mathbf{zero}\ ]\!]\ \rho \ \equiv\ \mathit{zero}$$

$$[\![\ \mathbf{succ}(n)\ ]\!]\ \rho \ \equiv\ \mathit{succ}(\ [\![\ n\ ]\!]\ \rho\ )$$

$$[\![\ \mathbf{lam}_A x \,.\, b(x)\ ]\!]\ \rho \ \equiv\ \lambda x \in A \,.\, [\![\ b(\mathbf{v}_i)\ ]\!]\ \rho[x/\mathbf{v}_i] \qquad\qquad (A \in PTYPE)$$

$$[\![\ \mathbf{pair}(a,b)\ ]\!]\ \rho \ \equiv\ \langle\ [\![\ a\ ]\!]\ \rho,\ [\![\ b\ ]\!]\ \rho\ \rangle$$

$$[\![\ \mathbf{inl}(a)\ ]\!]\ \rho \ \equiv\ \mathit{inl}([\![\ a\ ]\!]\ \rho)$$

$$[\![\ \mathbf{inr}(b)\ ]\!]\ \rho \ \equiv\ \mathit{inr}([\![\ b\ ]\!]\ \rho)$$

$$[\![\ \mathbf{nil}\ ]\!]\ \rho \ \equiv\ \mathit{nil}$$

$$[\![\ \mathbf{cons}(h,t)\ ]\!]\ \rho \ \equiv\ \mathit{cons}(\ [\![\ h\ ]\!]\ \rho,\ [\![\ t\ ]\!]\ \rho\ )$$

$$[\![\ \mathbf{cond}(b,t,u)\ ]\!]\ \rho \ \equiv\ \mathit{cond}(\ [\![\ b\ ]\!]\ \rho,\ [\![\ t\ ]\!]\ \rho,\ [\![\ u\ ]\!]\ \rho\ )$$

$$[\![\ \mathbf{natcase}(n,b,c)\ ]\!]\ \rho \ \equiv \\ \mathit{natcase}(\ [\![\ n\ ]\!]\ \rho,\ [\![\ b\ ]\!]\ \rho,\ \%x \,.\, [\![\ c(\mathbf{v}_i)\ ]\!]\ \rho[x/\mathbf{v}_i]\ )$$

$$[\![\ \mathbf{app}(f,a)\ ]\!]\ \rho \ \equiv\ [\![\ f\ ]\!]\ \rho\ `\ [\![\ a\ ]\!]\ \rho$$

$$[\![\ \mathbf{paircase}(p,b)\ ]\!]\ \rho \ \equiv \\ \mathit{paircase}(\ [\![\ p\ ]\!]\ \rho,\ \%x\ y \,.\, [\![\ c(\langle \mathbf{v}_i, \mathbf{v}_j\rangle)\ ]\!]\ \rho[x/\mathbf{v}_i][y/\mathbf{v}_j]\ )$$

$$[\![\ \mathbf{pluscase}(a,c,d)\ ]\!]\ \rho\ \equiv$$
$$\quad\quad\quad\quad \mathit{pluscase}(\ [\![\ a\ ]\!]\ \rho,\ \%x\ .\ [\![\ c(\mathbf{v}_i)\ ]\!]\ \rho[x/\mathbf{v}_i],\ \%x\ .\ [\![\ c(\mathbf{v}_i)\ ]\!]\ \rho[x/\mathbf{v}_i]\ )$$

$$[\![\ \mathbf{listcase}(l,b,c)\ ]\!]\ \rho\ \equiv$$
$$\quad\quad\quad\quad \mathit{listcase}(\ [\![\ l\ ]\!]\ \rho,\ [\![\ b\ ]\!]\ \rho,\ \%h\ t\ .\ [\![\ c(\mathbf{v}_i,\mathbf{v}_j)\ ]\!]\ \rho[h/\mathbf{v}_i][t/\mathbf{v}_j]\ )$$

$$[\![\ \mathbf{wrec}_{\prec}(h,a)\ ]\!]\ \rho\ \equiv\ \mathit{wrec}(\ \prec,\ \%x\ g\ .\ [\![\ h(\mathbf{v}_i,\mathbf{v}_j)\ ]\!]\ \rho[x/\mathbf{v}_i][g/\mathbf{v}_j],\ [\![\ a\ ]\!]\ \rho)$$

$$[\![\ \mathbf{eq}\ ]\!]\ \rho\ \equiv\ \Pi a \in A\,.\,\Pi b \in A\,.\,[x \in \mathit{bool}, a = b \leftrightarrow x = \mathit{true}] \qquad (A \in \mathit{ETYPE})$$

$$[\![\ \mathbf{v}_i\ ]\!]\ \rho\ \equiv\ \rho(\mathbf{v}_i)$$

References

[1] Peter Aczel. Frege structures and the notions of proposition, truth and set. In Keisler Barwise and Kunen, editors, *The Kleene Symposium*, pages 31–54. North Holland, 1980.

[2] Peter Dybjer. Program verification in a logical theory of constructions. Technical Report 26, Programming Methodology Group, Chalmers University of Technology, June 1986.

[3] C. A. R. Hoare and J. C. Shepherdson, editors. *Mathematical Logic and Programming Languages*. Prentice-Hall International, 1985.

[4] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[5] Per Martin-Löf. Constructive mathematics and computer programming. In Hoare and Shepherdson [3], pages 167–184, 1985.

[6] P. A. J. Noël. Experimenting with Isabelle in ZF set theory. Technical Report 177, University of Cambridge Computer Laboratory, September 1989.

[7] B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.

[8] B. Nordström and K. Petersson. Types and specifications. *Information Processing 83*, pages 915–920, 1983.

[9] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[10] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge Computer Laboratory, 1990.

[11] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *LICS '88, Edinburgh*, pages 384–391, 1988.

[12] Patrick Suppes. *Axiomatic Set Theory*. Dover, 1972.

A Proof of Strong Normalization For the Theory of Constructions Using a Kripke-Like Interpretation[1]

Thierry Coquand *Jean Gallier*[2]

INRIA
Domaine de Voluceau, Rocquencourt
B.P. 105
78153 Le Chesnay Cedex, France

Department of Computer and Information Science
University of Pennsylvania
200 South 33rd St.
Philadelphia, PA 19104, USA

Abstract. We give a proof that all terms that type-check in the theory of contructions are strongly normalizing (under β-reduction). The main novelty of this proof is that it uses a "Kripke-like" interpretation of the types and kinds, and that it does not use infinite contexts. We explore some consequences of strong normalization, consistency and decidability of type-checking. We also show that our proof yields another proof of strong normalization for LF (under β-reduction), using the reducibility method.

A Introduction

We give a proof that all terms that type-check in the theory of constructions are strongly normalizing (under β-reduction). The main novelty of this proof is that it uses a "Kripke-like" interpretation of the types and kinds, and that it does not use infinite contexts. The idea used for avoiding infinite contexts comes from Coquand's thesis [2]. Our proof yields as a corollary another proof of strong normalization (under β-reduction) of well-formed terms of LF. In fact, it is easy to see that this proof does not use the candidates of reducibility at all. We are unaware of similar proofs (using reducibility "à la Tait") for LF.

Our experience with proofs of strong normalization is that besides their intrinsic difficulty, their clarity and ease of understanding are greatly affected by the choice of notation, and the

[1]The results in this paper were first presented at the First Annual Workshop on Logical Frameworks, Esprit Basic Research Action, Antibes, May 7-11, 1990.

[2]Partially supported by ONR Grant NOOO14-88-K-0593.

order in which the concepts are introduced. For example, it is logical to define "the values" before defining $[\![\Gamma \triangleright A]\!]\rho\Delta$, the interpretation of types, since this latter definition requires the former (the term "values" is used by Coquand [3], but in Girard's terminology and ours, these are the sets $\mathcal{C}_{A,\Delta}$ of "candidates of reducibility"). However, we believe that it more intuitive and easier for understanding such proofs, if $[\![\Gamma \triangleright A]\!]\rho\Delta$ is defined *before* the families of candidates (In Coquand [3], $[\![\Gamma \triangleright A]\!]\rho\Delta$ is called the *interpretation of a term*, and it is denoted as $Eval\rho\ M$). It is possible to do so by first giving a rough and intuitive idea of what families of candidates are. Another difficulty is to package in a convenient way the various ingredients making up a candidate assignment (the substitution component, the candidate assignment component, etc). This is one place where the idea of viewing a context Δ as a world, as in Kripke semantics of intuitionistic logic, seems helpful.

A key remark for our presentation is the following: proofs of normalization that follow the reducibility method are intuitionistic. Hence, it should be possible to carry them in any intuitionistic model, hence in any Kripke model. There is furthermore one natural Kripke "term model" where we take for the Kripke worlds, the valid contexts of the type system. It should be noted that this paper represents "ongoing research", and that this is a preliminary version of a paper in which we intend to explore more thoroughly the nature of Kripke models for the theory of constructions.

Among the sources of inspiration for this research, Moggi and Mitchell's work on Kripke models for the simply-typed λ-calculus [1] should be mentioned. We also became recently aware of work by Aarne Ranta [5], in which the notion of "contexts as (finite approximation of) worlds" is used. One of the motivations for this work is to give an intuitionistic treatment of the notion of "possible worlds". Ranta's notion of Kripke structure is more general than ours, in that he does consider any *interpretation* between contexts, and not only projection (here seen dually as inclusion). It may be interesting to see if one can formulate a normalization proof in this framework.

One should be careful in referring to "the" theory of constructions, since different versions of this theory have been formulated and these versions are not all equivalent. Thus, in order to avoid ambiguities, we formulate in the next section a version of the theory of constructions equivalent (but not identical in syntax) to the version presented by Coquand and Huet [4]. We refer to this version of the theory of constructions as CC.

B Syntax of the Theory of Constructions CC

We find it pedagogically convenient to first describe a theory of constructions whose syntax has three levels (*kinds*, *type families*, and *terms*). The special kind $\star$ is the logical kind of propositions. In other words, types (propositions) are exactly those type families whose kind is $\star$. In the simple theory of type, Church used the notation o for $\star$. Another notation used in place of $\star$ is $Prop$ (or even $Type$). Furthermore, some authors use $Type$ instead of $kind$, but we find this practice somewhat confusing, since in the Curry-Howard formula–as–type analogy, propositions correspond to types.

We begin by defining *raw terms*.

DEFINITION B.1 We use the nonterminal K to range over kinds, A to range over type families, and M to range over terms. We also use two kinds of variables, ranging over kinds and type families. *Raw terms* are defined by the following grammar.

$$K \longrightarrow \star \mid (\Pi t\colon K)K \mid (\Pi x\colon A)K$$

$$A \longrightarrow t \mid (\forall t\colon K)A \mid (\forall x\colon A)A \mid (AA) \mid (AM) \mid (\Lambda t\colon K.\, A) \mid (\lambda x\colon A.\, A)$$

$$M \longrightarrow x \mid (MM) \mid (MA) \mid (\Lambda t\colon K.\, M) \mid (\lambda x\colon A.\, M).$$

A *context* is an ordered sequence of pairs $\Delta = \langle\langle x_1, A_1\rangle, \ldots, \langle x_m, A_m\rangle\rangle$, where x_i is a variable and A_i is a kind or a type family, and for any two x_i, A_i and x_j, A_j in Δ, $i \neq j$ implies that $x_i \neq x_j$. A context Δ is usually written as $x_1\colon A_1, \ldots, x_m\colon A_m$. There are four categories of *judgments*:

DEFINITION B.2 *Judgments* are expressions of the form:

$\Delta \triangleright$ (Δ is a valid context),

$\Delta \triangleright K\colon kind$ (K is a valid kind in context Δ),

$\Delta \triangleright A\colon K$ (A kind-checks with kind K in context Δ),

$\Delta \triangleright M\colon A$ (M type-checks with type A in context Δ).

We define β-reduction and β-conversion in the usual manner on raw terms. This means that redexes will be of the form $(\Lambda t\colon K.\, A)B$, $(\lambda x\colon A.\, B)M$, $(\Lambda t\colon K.\, M)B$, and $(\lambda x\colon A.\, M)N$. We emphasize that we *do not* consider η-conversion in this paper. There appears to be some difficulties with the Church-Rosser theorem if $\beta\eta$-conversion is defined on raw terms, and it seems that one needs to define judgments of the form $\Delta \triangleright M \stackrel{*}{\longleftrightarrow}_{CC} M'$ (equality judgments), which is quite cumbersome.

C Typing Rules for CC

We could list the typing rules assuming the above syntax, but it is possible to state them more concisely if certain conventions are adopted.

- Firstly, we will not distinguish between type variables and term variables.
- Secondly, we will use the symbol κ to denote either *kind* or $\star$.
- Thirdly, we will denote both judgments $\Delta \triangleright K\colon kind$ and $\Delta \triangleright \sigma\colon \star$ as $\Delta \triangleright A\colon \kappa$, and similarly, we will denote both judgments $\Delta \triangleright \sigma\colon K$ and $\Delta \triangleright M\colon \sigma$ as $\Delta \triangleright M\colon A$.
- Finally, we identify $\forall$ and Π, and Λ and λ.

Note that now, there is only one kind of raw terms given by the following grammar:

$M \longrightarrow x \mid \star \mid (\Pi x{:}\, M)M \mid (MM) \mid (\lambda x{:}\, M.\, M).$

With the above conventions, we only have one rule for each kind of rule.

DEFINITION C.1 In the rules below, $\kappa, \kappa_1, \kappa_2 \in \{\star, kind\}$.

Context Formation:

$$\emptyset \rhd \qquad \text{empty context}$$

$$\frac{\Delta \rhd}{\Delta \rhd \star{:}\, kind}$$

$$\frac{\Delta \rhd A{:}\, \kappa}{\Delta, x{:}\, A \rhd} \qquad x \notin dom(\Delta)$$

Axiomatic Judgments:

$$\frac{\Delta \rhd}{\Delta \rhd x{:}\, A} \qquad x{:}\, A \in \Delta$$

Product Formation and Quantification:

$$\frac{\Delta \rhd A_1{:}\, \kappa_1 \quad \Delta, x{:}\, A_1 \rhd A_2{:}\, \kappa_2}{\Delta \rhd (\Pi x{:}\, A_1)A_2{:}\, \kappa_2}$$

Abstraction:

$$\frac{\Delta \rhd A_1{:}\, \kappa_1 \quad \Delta, x{:}\, A_1 \rhd A_2{:}\, \kappa_2 \quad \Delta, x{:}\, A_1 \rhd M{:}\, A_2}{\Delta \rhd (\lambda x{:}\, A_1.\, M){:}\, (\Pi x{:}\, A_1)A_2}$$

Application:

$$\frac{\Delta \rhd M{:}\, (\Pi x{:}\, A_1)A_2 \quad \Delta \rhd N{:}\, A_1}{\Delta \rhd MN{:}\, A_2[N/x]}$$

Kind and Type Conversion:

$$\frac{\Delta \rhd M{:}\, A_1 \quad \Delta \rhd A_2{:}\, \kappa \quad A_1 \overset{*}{\longleftrightarrow}_{CC} A_2}{\Delta \rhd M{:}\, A_2}$$

It turns out that the above typing rules can be simplified, because some of the premises are redundant. Of course, this has to be justified carefully, but this has been verified by Coquand and Huet [4], and others. For the reader's convenience, we recall some of the main basic properties of CC.

D Some Basic Properties of CC

We shall use the notation $\Delta \triangleright E$ as an abbreviation for all forms of judgments. Given contexts Γ and Δ, the notation $\Gamma \leq \Delta$ means that Γ is an initial subsequence of Δ.

First, we note that under β-conversion alone, the Church-Rosser theorem holds even for raw terms.

THEOREM D.1 *(Martin Löf)*
The Church-Rosser property holds for raw terms of CC (even the economical version).

Proof. Such a proof using the so called "Tait/Martin Löf's method" was given by Martin Löf [6]. $\square$

It should be noted that theorem D.1 is quite handy. It appears that if β-conversion is defined on raw terms, which is definitely more convenient than using equality judgments, many important properties of CC make use of the Church-Rosser property.

The propositions listed below consist of the translation in English and in our terminology of properties 1-7 in Chapter 1 of Coquand's thesis [2]. In some cases, these proofs require some amplification. First, we need the following definitions, which are translations in our terminology of Coquand's definitions.

DEFINITION D.1 K is a *kind* iff K is of the form $\star$ or $(\Pi x_1\colon A_1)\ldots(\Pi x_m\colon A_m)\star$, and $\Delta \triangleright K\colon kind$ for some context Δ;

A is a *type family* iff $\Delta \triangleright A\colon K$ for some context Δ and some kind K;

A is a *type* iff $\Delta \triangleright A\colon\star$ for some context Δ;

M is a *proof* (or *proof term*) iff $\Delta \triangleright M\colon A$ where A is not a kind.

When we want to stress that a context Δ is well-formed, that is, when $\Delta \triangleright$ is derivable, we say that Δ is a *valid context*, and similarly for kinds, type families, types, and proofs.

LEMMA D.2 *If $\Delta \triangleright E$, then $\Delta' \triangleright$ for every $\Delta' \leq \Delta$, and more generally, every derivation of $\Delta \triangleright E$ contains a derivation of $\Delta' \triangleright$ as a subderivation.*

LEMMA D.3 *If $\Delta \triangleright A\colon K$ and $\Delta \triangleright A\colon K'$ where both K and K' are kinds, then $K \overset{*}{\longleftrightarrow}_{CC} K'$. Similarly, if $\Delta \triangleright M\colon A$ and $\Delta \triangleright M\colon A'$, where both A and A' are not kinds, then $A \overset{*}{\longleftrightarrow}_{CC} A'$.*

The proof of the above lemma actually seems to require the Church-Rosser property and the following proposition.

PROPOSITION D.4 *Assume that $\Delta \triangleright (\Pi x\colon A)K\colon kind$ and $\Delta \triangleright (\Pi x\colon A')K'\colon kind$. If $(\Pi x\colon A)K \overset{*}{\longleftrightarrow}_{CC} (\Pi x\colon A')K'$, then $A \overset{*}{\longleftrightarrow}_{CC} A'$ and $K \overset{*}{\longleftrightarrow}_{CC} K'$.*

PROPOSITION D.5 *Both $\Delta \triangleright M\colon A$ and $\Delta \triangleright M\colon kind$ are not derivable at the same time.*

DEFINITION D.6 Given any two contexts Δ, Δ', we say that $\Delta \subseteq \Delta'$ iff for every x, if $x \in dom(\Delta)$ then $x \in dom(\Delta')$ and $\Delta(x) = \Delta'(x)$.

LEMMA D.7 *Assume that* $\Delta \triangleright$, $\Delta' \triangleright$, *and* $\Delta \subseteq \Delta'$. *If* $\Delta \triangleright E$, *then* $\Delta' \triangleright E$. *In particular, if* $\Delta \leq \Delta'$, $\Delta' \triangleright$, *and* $\Delta \triangleright E$, *then* $\Delta' \triangleright E$.

LEMMA D.8 *If* $\Delta \triangleright M\colon A$ *and* $\Delta, x\colon A, \Delta' \triangleright E$, *then* $\Delta, \Delta'[M/x] \triangleright E[M/x]$.

LEMMA D.9 *If* $\Delta \triangleright M\colon A$ *and* A *is not a kind, then* $\Delta \triangleright A\colon\star$. *If* $\Delta \triangleright M\colon A$ *and* A *is a kind, then* $\Delta \triangleright A\colon kind$.

LEMMA D.10 *If* $\Delta, x\colon A, \Delta' \triangleright E$, $A \xleftrightarrow{*}_{CC} A'$, *and either* $\Delta \triangleright A'\colon\star$ *or* $\Delta \triangleright A'\colon kind$, *then* $\Delta, x\colon A', \Delta' \triangleright E$.

LEMMA D.11 *If* $\Delta \triangleright M\colon A$ *and* $M \xrightarrow{*}_{CC} N$, *then* $\Delta \triangleright N\colon A$.

LEMMA D.12 *The judgments* $\Delta \triangleright M\colon A$ *where* A *is a type and* $\Delta \triangleright M\colon K$ *where* K *is a kind cannot hold simultaneously.*

The proof of the above above lemma seems to require the Church-Rosser property.

In view of proposition D.5, kinds are disjoint from type families and proofs. In view of lemma D.12, proofs and type families are disjoint.

E Strong Normalization in CC

The proof of strong normalization for well-typed terms of CC is obtained by generalizing the proof given by Girard for the system F_ω [13], as presented in Gallier [12]. However, there are some significant technical complications. In F_ω, we have an ascending hierarchy, kinds, type families, and terms, where kinds do not depend on type families or terms, and type families do not depend on terms. However, in CC, kinds, type families, and terms, are defined in a single big simultaneous inductive construction. The main difficulty is to ensure that the interpretations $[\![\Gamma \triangleright A]\!]\rho\Delta$ are nondegenerate (i.e., nonempty sets).

The first step is to define the concept of a candidate assignment, which packages together a substitution and a valuation assigning candidates to variables.

DEFINITION E.1 A *substitution* is a function $\varphi\colon \mathcal{V} \to Terms$ such that $\varphi(x) \neq x$ for only finitely many x, and for every $\varphi(x)$, there is some context Δ and either some type A such that $\Delta \triangleright \varphi(x)\colon A$ (and $\Delta \triangleright A\colon\star$), or some kind K such that $\Delta \triangleright \varphi(x)\colon K$ (and $\Delta \triangleright K\colon kind$). The domain $D(\varphi)$ of φ is the set $D(\varphi) = \{x \mid \varphi(x) \neq x\}$.

Every substitution φ has a unique homomorphic extension $\widehat{\varphi}\colon Terms \to Terms$. Given a term M (term, type family, or kind), the result of applying φ to M is $\widehat{\varphi}(M)$, and it is denoted as $\varphi(M)$ or $M[\varphi]$.

Some form of Kripke structure is lurking around. Contexts are going to play the role of worlds. Consequently, most concepts will be defined "in world Δ". The notion of inclusion of worlds is the relation $\subseteq$ defined in definition D.6. Substitutions will also play the role of valuations assigning values to variables. This motivates the following definition.

Definition E.2 Given two valid contexts Γ, Δ, where Γ is used to type/kind check, and Δ acts as a world, given a substitution φ, we say that $\Gamma[\varphi]$ *type-checks in* Δ iff $\Delta \triangleright x[\varphi] \colon \Gamma(x)[\varphi]$ for every $x \in dom(\Gamma)$.

At first glance, one may be concerned that this condition is circular. However, this is not so. Indeed, if $\Gamma = x_1 \colon A_1, \ldots, x_m \colon A_m$ is a valid context, it can be easily shown that $FV(A_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for all i, $1 \leq i \leq n$, and that $\Gamma[\varphi]$ type-checks in Δ means that $\Delta \triangleright x_i[\varphi] \colon A_i[x_1[\varphi]/x_1, \ldots, x_{i-1}[\varphi]/x_{i-1}]$ for all i, $1 \leq i \leq n$, which is possible.

We now assume that for every world Δ and type family A that kind-checks in Δ, we have a set $\mathcal{C}_{A,\Delta}$ of nonempty sets called *candidates* to be defined soon. All we need to know is that, when A is a type, every $C \in \mathcal{C}_{A,\Delta}$ is a set of terms $\Delta' \triangleright M$ such that $\Delta' \triangleright M \colon A$ for some $\Delta' \supseteq \Delta$, and when A kind-checks with kind $(\Pi x \colon B)K$, every element of $\mathcal{C}_{A,\Delta}$ is a certain function. We also have a set $\mathcal{C}_{\star,\Delta}$ consisting of nonempty sets of types $\Delta' \triangleright A$ such that $\Delta' \triangleright A \colon \star$ for some $\Delta' \supseteq \Delta$, and a set $\mathcal{C}_{kind,\Delta}$ consisting of nonempty sets of kinds $\Delta' \triangleright K$ such that $\Delta' \triangleright K \colon kind$ for some $\Delta' \supseteq \Delta$.

Definition E.3 A *candidate assignment* is any function ρ from $\mathcal{V} \cup \{\star, kind\}$ to $Terms \cup (Terms \times \bigcup \mathcal{C})$, such that the following properties hold:

(1) If we define the function $\rho_s \colon \mathcal{V} \to Terms$ such that,

$$\rho_s(x) = \begin{cases} A & \text{if } \rho(x) = \langle A, C \rangle, \\ A & \text{if } \rho(x) = A, \end{cases}$$

then ρ_s is a substitution (which means that $\rho_s(x) \neq x$ only for finitely many $x \in \mathcal{V}$), and,

(2) If $\rho(x) = \langle A, C \rangle$, then A is a type-family that kind-checks in some context Δ and $C \in \mathcal{C}_{A,\Delta}$, else if $\rho(x) = A$ then A is a term (proof) that type-checks in some context Δ;

(3) $\rho(\star) = \langle \star, C \rangle$, $C \in \mathcal{C}_{\star,\Delta}$, $\rho(kind) = \langle kind, C \rangle$, and $C \in \mathcal{C}_{kind,\Delta}$.

The function ρ also defines another function ρ_c such that $x \mapsto C$, $\star \mapsto C$, and $kind \mapsto C$. By abuse of notation, both ρ_s and ρ_c are often denoted as ρ, when the context makes it clear which is referred to.

Definition E.4 A candidate assignment ρ *satisfies* Γ *at* Δ iff

(1) $\Gamma[\rho_s]$ type-checks in Δ.

(2) Whenever $\rho(x) = \langle A, C \rangle$ or $\rho(x) = A$, then A kind/type-checks in Δ, and $C \in \mathcal{C}_{A,\Delta}$.

It is easy to verify that if $\Delta \subseteq \Delta'$ and ρ satisfies Γ at Δ, then ρ satisfies Γ at Δ'. We can now define $[\![\Gamma \triangleright A]\!]\rho\Delta$, where Γ is a context, either A is a type family that kind-ckecks in Γ, or A is a kind valid in Γ, or $A = \mathit{kind}$, ρ is a candidate assignment, and Δ is a context viewed as a world. The definition is by induction on the complexity of $\Gamma \triangleright A$ (if $\Gamma = x_1 : A_1, \ldots, x_m : A_m$, then the complexity of $\Gamma \triangleright A$ is the sum of the sizes of $A_1, \ldots, A_m, A$). It only makes sense when ρ satisfies Γ at Δ.

DEFINITION E.5 In the clauses below, K stands for a kind, σ for a type, A, B for type families, D for a kind or a type, M for a type family or a term (proof), and N for a term (proof).

$$
\begin{aligned}
[\![\Gamma \triangleright \mathit{kind}]\!]\rho\Delta &= \rho_c(\mathit{kind}), \\
[\![\Gamma \triangleright \star]\!]\rho\Delta &= \rho_c(\star), \\
[\![\Gamma \triangleright x]\!]\rho\Delta &= \rho_c(x), \\
[\![\Gamma \triangleright AB]\!]\rho\Delta &= [\![\Gamma \triangleright A]\!]\rho\Delta(\Delta \triangleright B[\rho_s],\ [\![\Gamma \triangleright B]\!]\rho\Delta), \\
[\![\Gamma \triangleright AN]\!]\rho\Delta &= [\![\Gamma \triangleright A]\!]\rho\Delta(\Delta \triangleright N[\rho_s]), \\
[\![\Gamma \triangleright (\Pi x : K)D]\!]\rho\Delta &= \{\Delta' \triangleright M \mid \Delta' \triangleright M : ((\Pi x : K)D)[\rho_s],\ \Delta' \supseteq \Delta, \text{ and} \\
&\qquad \forall \Delta'' \supseteq \Delta',\ \forall \Delta'' \triangleright A \in [\![\Gamma \triangleright K]\!]\rho\Delta'',\ \forall C \in \mathcal{C}_{A,\Delta''}, \\
&\qquad \Delta'' \triangleright (MA) \in [\![\Gamma, x : K \triangleright D]\!]\rho[x := \langle A, C\rangle]\Delta''\}, \\
[\![\Gamma \triangleright (\Pi x : \sigma)D]\!]\rho\Delta &= \{\Delta' \triangleright M \mid \Delta' \triangleright M : ((\Pi x : \sigma)D)[\rho_s],\ \Delta' \supseteq \Delta, \text{ and} \\
&\qquad \forall \Delta'' \supseteq \Delta',\ \forall \Delta'' \triangleright N \in [\![\Gamma \triangleright \sigma]\!]\rho\Delta'', \\
&\qquad \Delta'' \triangleright (MN) \in [\![\Gamma, x : \sigma \triangleright D]\!]\rho[x := N]\Delta''\}, \\
[\![\Gamma \triangleright \lambda x : K.\, B]\!]\rho\Delta &= \lambda(\Delta' \triangleright A)\lambda C.\, [\![\Gamma, x : K \triangleright B]\!]\rho[x := \langle A, C\rangle]\Delta', \\
&\qquad \text{a function with domain} \\
&\qquad \{\langle \Delta' \triangleright A, C\rangle \mid \Delta' \triangleright A : K[\rho_s],\ \Delta' \supseteq \Delta,\ C \in \mathcal{C}_{A,\Delta'}\}, \\
[\![\Gamma \triangleright \lambda x : \sigma.\, B]\!]\rho\Delta &= \lambda(\Delta' \triangleright N).\, [\![\Gamma, x : \sigma \triangleright B]\!]\rho[x := N]\Delta', \\
&\qquad \text{a function with domain} \\
&\qquad \{\Delta' \triangleright N \mid \Delta' \triangleright N : \sigma[\rho_s],\ \Delta' \supseteq \Delta\}.
\end{aligned}
$$

We emphasize again the fact that in $[\![\Gamma \triangleright x]\!]\rho\Delta$, we have $\Gamma \triangleright x : K$ for some kind K, i.e., x is a type variable.

DEFINITION E.6 Given any judgment $\Gamma \triangleright M : A$ (where A can even be *kind*), given any candidate assignment ρ, and any context Δ viewed as a world, we write $\Delta \vdash \Gamma[\rho]$ iff

(1a) ρ satisfies Γ at Δ, and

(2a) $\Delta \triangleright x[\rho] \in [\![\Gamma \triangleright \Gamma(x)]\!]\rho\Delta$ for every $x \in \mathit{dom}(\Gamma)$.

We will also write $\Delta \vdash (M : A)[\rho]$ iff

(1b) ρ satisfies Γ at Δ, and

(2b) $\Delta \triangleright M[\rho] \in [\![\Gamma \triangleright A]\!]\rho\Delta$.

Then, the main theorem reads as follows: Whenever $\Gamma \triangleright M: A$ and $\Delta \vdash \Gamma[\rho]$, then $\Delta \vdash (M: A)[\rho]$. This looks like a Kripke-style type soundness result.

Actually, it is not obvious that the inductive definition of $[\![\Gamma \triangleright A]\!]\rho\Delta$ defines nonempty sets and total functions, and this depends on some properties of the sets $\mathcal{C}_{A,\Delta}$. One of the crucial facts is that for every valid context Δ and type or kind A, there is some term or type family $\Delta' \triangleright M$ with $\Delta' \supseteq \Delta$ such that $\Delta' \triangleright M: A$. Indeed $\Delta' = \Delta, x: A$ where $x \notin dom(\Delta)$ does the job, since $\Delta, x: A \triangleright x: A$ is derivable.

We can now define the sets $\mathcal{C}_{A,\Delta}$. For this this, we need a complexity measure for types and kinds.

Definition E.7 Let A be any valid type, and K any valid kind. We define $c(A)$ and $c(K)$ inductively as follows:

$$c(A) = 0,$$

$$c(K) = \begin{cases} 1 & \text{if } K = \star, \\ max(c(B), c(D)) + 1 & \text{if } K = (\Pi x: B)D. \end{cases}$$

It is easily verified that if $K \overset{*}{\longleftrightarrow}_{CC} K'$, then $c(K) = c(K')$. The main property of this complexity measure is that it is invariant under substitution.

Lemma E.8 *For every type family or term M, for every kind K, $c(K[M/x]) = c(K)$.*

Proof. We proceed by induction on the structure of K. If $K = \star$, the lemma holds since $\star[M/x] = \star$. If $K = (\Pi x: B)D$, there are two cases. If B is also a kind, by the induction hypothesis, $c(B[M/x]) = c(B)$, $c(D[M/x]) = c(D)$, and the lemma holds since $K[M/x] = (\Pi x: B[M/x])D[M/x]$. If A is a type, then $A[M/x]$ is also a type, and since $c(A[M/x]) = 0$ and by the induction hypothesis $c(D[M/x]) = c(D)$, the lemma holds. $\square$

We also let $c(kind) = 0$. The sets $\mathcal{C}_{A,\Delta}$ are defined by induction on $c(K)$, where $\Delta \triangleright A: K$. Since $c(K)$ only depends on the equivalence class of K modulo β-conversion, this definition is proper. The definition of the sets $\mathcal{C}_{A,\Delta}$ given next is a bit more general than really required for proving strong normalization. The reason for giving it in this form is that it can be used to extend our proof to other properties besides strong normalization. This definition also contains all the closure conditions that will come up during the proof of the main result.

Definition E.9 The family $\mathcal{C}$ of sets $\mathcal{C}_{A,\Delta}$ where A is a kind or a type family valid in the context Δ, is defined by the properties listed below. It is called the *family of saturated sets*.

1. $\mathcal{C}_{kind,\Delta}$ is the set of sets C, such that, each C is a nonempty set of strongly normalizing kinds $\Delta' \triangleright K$, with $\Delta' \supseteq \Delta$, and the following properties hold:

   (a) $\Delta' \triangleright \star \in C$ for all $\Delta' \supseteq \Delta$.

   (b) For every kind $\Delta' \triangleright (\Pi x: K)D$, with $\Delta' \supseteq \Delta$ and K a kind, if $\Delta' \triangleright K \in C$ and $\Delta' \triangleright D \in C$, then $\Delta' \triangleright (\Pi x: K)D \in C$.

(c) For every kind $\Delta' \triangleright (\Pi x{:}\, \sigma)D$, with $\Delta' \supseteq \Delta$ and σ a type, for every $C' \in \mathcal{C}_{\star,\Delta}$, if $\Delta' \triangleright \sigma \in C'$ and $\Delta' \triangleright D \in C$, then $\Delta' \triangleright (\Pi x{:}\, \sigma)D \in C$.

(d) Whenever $\Delta' \triangleright K \in C$ and $\Delta' \subseteq \Delta''$, then $\Delta'' \triangleright K \in C$.

2. $\mathcal{C}_{\star,\Delta}$ is the set of sets C, such that, each C is a nonempty set of strongly normalizing types $\Delta' \triangleright A$, with $\Delta' \supseteq \Delta$, and the following properties hold:

(S0) For every type $\Delta' \triangleright (\Pi x{:}\, K)A$, with $\Delta' \supseteq \Delta$ and K a kind, for every $C' \in \mathcal{C}_{kind,\Delta}$, if $\Delta' \triangleright K \in C'$ and $\Delta' \triangleright A \in C$, then $\Delta' \triangleright (\Pi x{:}\, K)A \in C$, and

For every type $\Delta' \triangleright (\Pi x{:}\, \sigma)A$, with $\Delta' \supseteq \Delta$ and σ a type, if $\Delta' \triangleright \sigma \in C$ and $\Delta' \triangleright A \in C$, then $\Delta' \triangleright (\Pi x{:}\, \sigma)A \in C$.

(S1) For every variable x, if $\Delta' \triangleright xN_1 \ldots N_m{:}\star$ for some $\Delta' \supseteq \Delta$ and $N_1, \ldots, N_m$ are SN, then $\Delta' \triangleright xN_1 \ldots N_m \in C$.

(S2) Whenever $\Delta' \triangleright M[N/x]N_1 \ldots N_m{:}\star$ and $\Delta' \triangleright N{:}\, B$ is SN for some $\Delta' \supseteq \Delta$, if $\Delta' \triangleright M[N/x]N_1 \ldots N_m \in C$, then $\Delta' \triangleright (\lambda x{:}\, B.\, M)NN_1 \ldots N_m \in C$.

(S3) Whenever $\Delta' \triangleright A \in C$ and $\Delta' \subseteq \Delta''$, then $\Delta'' \triangleright A \in C$.

3. When A is a type (and $\Delta \triangleright A{:}\star$), $\mathcal{C}_{A,\Delta}$ is the set of sets C, such that, each C is a nonempty set of strongly normalizing terms $\Delta' \triangleright M$ such that $\Delta' \triangleright M{:}\, A$ for some $\Delta' \supseteq \Delta$, and the following properties hold:

(S1) For every variable x, if $\Delta' \triangleright xN_1 \ldots N_m{:}\, A$ for some $\Delta' \supseteq \Delta$ and $N_1, \ldots, N_m$ are SN, then $\Delta' \triangleright xN_1 \ldots N_m \in C$.

(S2) Whenever $\Delta' \triangleright M[N/x]N_1 \ldots N_m{:}\, A$ and $\Delta' \triangleright N{:}\, B$ is SN for some $\Delta' \supseteq \Delta$, if $\Delta' \triangleright M[N/x]N_1 \ldots N_m \in C$, then $\Delta' \triangleright (\lambda x{:}\, B.\, M)NN_1 \ldots N_m \in C$.

(S3) Whenever $\Delta' \triangleright M \in C$ and $\Delta' \subseteq \Delta''$, then $\Delta'' \triangleright M \in C$.

4. When A is a type family such that $\Delta \triangleright A{:}\, (\Pi x{:}\, B)D$ (and $\Delta \triangleright (\Pi x{:}\, B)D{:}\, kind$), $\mathcal{C}_{A,\Delta}$ is the set of functions with the following properties:

(a) If B is a kind, then

- $f \in \mathcal{C}_{A,\Delta}$ is a function with domain

$$\{\langle \Delta' \triangleright M, C \rangle) \mid \Delta' \triangleright M{:}\, B,\ \Delta' \supseteq \Delta,\ C \in \mathcal{C}_{M,\Delta'}\}$$

such that $f(\Delta' \triangleright M, C) \in \mathcal{C}_{AM,\Delta'}$, and

- $f(\Delta' \triangleright M_1, C) = f(\Delta' \triangleright M_2, C)$ whenever $M_1 \overset{*}{\longleftrightarrow}_{CC} M_2$.

(b) If B is a type, then

- $f \in \mathcal{C}_{A,\Delta}$ is a function with domain $\{\Delta' \triangleright N \mid \Delta' \triangleright N\colon B,\ \Delta' \supseteq \Delta\}$ such that $f(\Delta' \triangleright N) \in \mathcal{C}_{AN,\Delta'}$, and

- $f(\Delta' \triangleright N_1) = f(\Delta' \triangleright N_2)$ whenever $N_1 \stackrel{*}{\longleftrightarrow}_{CC} N_2$.

Note that this definition is proper, because we can prove that the sets $\mathcal{C}_{M,\Delta'}$, $\mathcal{C}_{AM,\Delta'}$, and $\mathcal{C}_{AN,\Delta'}$, needed in (4) are well defined, where $\Delta \triangleright A\colon(\Pi x\colon B)D$, $\Delta' \triangleright M\colon B$, and $\Delta' \triangleright N\colon B$ with $\Delta' \supseteq \Delta$. This is correct, since $\Delta' \triangleright AM\colon D[M/x]$, $\Delta' \triangleright AN\colon D[N/x]$, $c(B) < c((\Pi x\colon B)D)$, and by lemma E.8, $c(D[M/x]) = c(D) < c((\Pi x\colon B)D)$, and $c(D[N/x]) = c(D) < c((\Pi x\colon B)D)$. One can also easily prove that if $A \stackrel{*}{\longleftrightarrow}_{CC} A'$, then $\mathcal{C}_{A,\Delta} = \mathcal{C}_{A',\Delta}$.

Given a type family A such that $\Delta \triangleright A\colon K$, we can prove by induction on $c(K)$ that each $\mathcal{C}_{A,\Delta}$ is nonempty.

Lemma E.10 *Whenever A kind-checks in Δ, $\mathcal{C}_{A,\Delta}$ is nonempty.*

Proof. We define an element $can_{A,\Delta}$ of $\mathcal{C}_{A,\Delta}$ where $\Delta \triangleright A\colon K$ such that $A \stackrel{*}{\longleftrightarrow}_{CC} A'$ implies that $can_{A,\Delta} = can_{A',\Delta}$, by induction on $c(K)$. We call $can_{A,\Delta}$ the *canonical member* of $\mathcal{C}_{A,\Delta}$.

When $A = kind$, note that the set $can_{kind,\Delta}$ of strongly normalizing kinds of the form $\Delta' \triangleright K$ for some $\Delta' \supseteq \Delta$ is nonempty, since $\Delta' \triangleright \star\colon kind$ for every Δ', and it is obvious that (b), (c), and (d), are also satisfied.

When $A = \star$, note that the set $can_{\star,\Delta}$ of strongly normalizing types of the form $\Delta' \triangleright N$ for some $\Delta' \supseteq \Delta$ is nonempty, since $\Delta, x\colon\star \triangleright x\colon\star$ for $x \notin dom(\Delta)$. Properties (S0), (S1), (S2), and (S3), are also easily verified.

When A is a type, note that the set $can_{A,\Delta}$ of strongly normalizing terms of the form $\Delta' \triangleright N$ such that $\Delta' \triangleright N\colon A$ for some $\Delta' \supseteq \Delta$ is nonempty, since $\Delta, x\colon A \triangleright x\colon A$ for $x \notin dom(\Delta)$. Properties (S1), (S2), and (S3), are also easily verified. That $A \stackrel{*}{\longleftrightarrow}_{CC} A'$ implies $can_{A,\Delta} = can_{A',\Delta}$ follows from the fact that $\Delta' \triangleright N\colon A$ and $A \stackrel{*}{\longleftrightarrow}_{CC} A'$ implies that $\Delta' \triangleright N\colon A'$.

When $\Delta \triangleright A\colon(\Pi x\colon B)D$, we define the function $can_{A,\Delta}$ as follows. By the induction hypothesis, for every M such that $\Delta' \triangleright M\colon B$ for some $\Delta' \supseteq \Delta$, $can_{M,\Delta'}$ is defined. We define $can_{A,\Delta}$ such that $can_{A,\Delta}(\Delta' \triangleright M, C) = can_{AM,\Delta'}$, and $can_{A,\Delta}(\Delta' \triangleright M) = can_{AM,\Delta'}$ if B is a type. If $A \stackrel{*}{\longleftrightarrow}_{CC} A'$, then $AM \stackrel{*}{\longleftrightarrow}_{CC} A'M$, and this implies $can_{AM,\Delta'} = can_{A'M,\Delta'}$ by the induction hypothesis. $\square$

Remark: It will be observed later that for proving strong normalization, we can simply define $\mathcal{C}_{kind,\Delta}$ and $\mathcal{C}_{\star,\Delta}$ as the singleton families $\mathcal{C}_{kind,\Delta} = \{can_{kind,\Delta}\}$ and $\mathcal{C}_{\star,\Delta} = \{can_{\star,\Delta}\}$.

In order to show that the closure properties of the family $\mathcal{C}$ insure that the sets $[\![\Gamma \triangleright A]\!]\rho\Delta$ are also in $\mathcal{C}$, we need the following technical lemma.

Lemma E.11 *If $\mathcal{C}$ is the family of saturated sets, for any two ρ and ρ' satisfying Γ at Δ, if $x[\rho] \stackrel{*}{\longleftrightarrow}_{CC} x[\rho']$ for every $x \in dom(\Gamma)$, then $[\![\Gamma \triangleright A]\!]\rho\Delta = [\![\Gamma \triangleright A]\!]\rho'\Delta$.*

Proof. A fairly simple induction on the size of A. $\square$

Now, we can prove that $\mathcal{C}$ contains the sets $[\![\Gamma \triangleright A]\!]\rho\Delta$.

LEMMA E.12 *If $\mathcal{C}$ is the family of saturated sets, whenever ρ satisfies Γ at Δ, then* $[\![\Gamma \triangleright A]\!]\rho\Delta \in \mathcal{C}_{A[\rho],\Delta}$.

Proof. One proceeds by induction on the size of A, also adding to the induction hypothesis the fact proved in lemma E.11 that for any two ρ and ρ' satisfying Γ at Δ, if $x[\rho] \overset{*}{\longleftrightarrow}_{CC} x[\rho']$ for every $x \in dom(\Gamma)$, then $[\![\Gamma \triangleright A]\!]\rho\Delta = [\![\Gamma \triangleright A]\!]\rho'\Delta$. $\square$

Given two valid contexts $\Gamma = x_1 : A_1, \ldots, x_m : A_m$ and $\Gamma' = x_1 : A'_1, \ldots, x_m : A'_m$, we say that $\Gamma \overset{*}{\longleftrightarrow}_{CC} \Gamma'$ iff $A_i \overset{*}{\longleftrightarrow}_{CC} A'_i$ for all i, $1 \leq i \leq m$.

LEMMA E.13 *If $\mathcal{C}$ is the family of saturated sets, whenever ρ satisfies Γ and Γ' at Δ and $\Gamma \overset{*}{\longleftrightarrow}_{CC} \Gamma'$, then $[\![\Gamma \triangleright A]\!]\rho\Delta = [\![\Gamma' \triangleright A]\!]\rho\Delta$.*

Proof. A fairly simple induction on the size of A. $\square$

We also have the following technical property known as "substitution property". This is perhaps the lemma whose proof is the most technical.

LEMMA E.14 *If $\mathcal{C}$ is the family of saturated sets, and ρ satisfies Γ at Δ, if $\Gamma, x : K \triangleright A : B$ for some B, and $\Gamma \triangleright D : K$ where K is a kind, then*

$$[\![\Gamma \triangleright A[D/x]]\!]\rho\Delta = [\![\Gamma, x : K \triangleright A]\!]\rho[x := \langle D[\rho], [\![\Gamma \triangleright D]\!]\rho\Delta\rangle]\Delta,$$

and if $\Gamma, x : \sigma \triangleright A : B$ for some B, and $\Gamma \triangleright M : \sigma$ where σ is a type, then

$$[\![\Gamma \triangleright A[M/x]]\!]\rho\Delta = [\![\Gamma, x : \sigma \triangleright A]\!]\rho[x := M[\rho]]\Delta.$$

Proof. In order to prove this lemma, it is necessary to prove the following stronger property:

Assuming that ρ satisfies Γ, Γ' at Δ, if $\Gamma, x : K, \Gamma' \triangleright A : B$ for some B, and $\Gamma \triangleright D : K$ where K is a kind, then

$$[\![\Gamma, \Gamma'[D/x] \triangleright A[D/x]]\!]\rho\Delta = [\![\Gamma, x : K, \Gamma' \triangleright A]\!]\rho[x := \langle D[\rho], [\![\Gamma \triangleright D]\!]\rho\Delta\rangle]\Delta,$$

and if $\Gamma, x : \sigma, \Gamma' \triangleright A : B$ for some B, and $\Gamma \triangleright M : \sigma$ where σ is a type, then

$$[\![\Gamma, \Gamma'[M/x] \triangleright A[M/x]]\!]\rho\Delta = [\![\Gamma, x : \sigma, \Gamma' \triangleright A]\!]\rho[x := M[\rho]]\Delta.$$

The proof of this property is by induction on the size of A, and it uses lemma E.11 and lemma E.13. $\square$

Using the previous lemma, we can show the following important lemma.

LEMMA E.15 *If $\mathcal{C}$ is the family of saturated sets, whenever ρ satisfies Γ at Δ and $A \overset{*}{\longleftrightarrow}_{CC} A'$, then $[\![\Gamma \triangleright A]\!]\rho\Delta = [\![\Gamma \triangleright A']\!]\rho\Delta$.*

Proof. The proof is by induction on the sum of the sizes of A and A', and it uses lemma E.11, lemma E.13, and lemma E.14. $\square$

Finally, we can prove the main theorem. Recall from definition E.6 that $\Delta \vdash \Gamma[\rho]$ means

(1) ρ satisfies Γ at Δ, and

(2) $\Delta \triangleright x[\rho] \in [\![\Gamma \triangleright \Gamma(x)]\!]\rho\Delta$ for every $x \in dom(\Gamma)$.

It is easy to verify that if $\Delta \subseteq \Delta'$ and $\Delta \vdash \Gamma[\rho]$, then $\Delta' \vdash \Gamma[\rho]$.

THEOREM E.1 *If $\mathcal{C}$ is the family of saturated sets, whenever $\Gamma \triangleright M\colon A$ and $\Delta \vdash \Gamma[\rho]$, then $\Delta \triangleright M[\rho] \in [\![\Gamma \triangleright A]\!]\rho\Delta$.*

Proof. The proof is by induction on a deduction proving that A type/kind-checks in Γ. Lemma E.15 is crucial in taking care of the case where the last inference is the type or kind equality rule. $\square$

As mentioned earlier, if we define $\Delta \vdash (M\colon A)[\rho]$ iff

(1) ρ satisfies Γ at Δ, and

(2) $\Delta \triangleright M[\rho] \in [\![\Gamma \triangleright A]\!]\rho\Delta$,

then, the main theorem reads as follows:

Whenever $\Gamma \triangleright M\colon A$ and $\Delta \vdash \Gamma[\rho]$, then $\Delta \vdash (M\colon A)[\rho]$, and this looks like a Kripke-style type soundness result.

By letting $[\rho]$ be the identity substitution and ρ_c assign the canonical element $can_{\Gamma(x),\Gamma}$ to each $x \in dom(\Gamma)$, $can_{\star,\Gamma}$ to $\star$, and $can_{kind,\Gamma}$ to *kind*, we obtain the fact that all valid terms of the theory of construction are SN.

THEOREM E.2 *Whenever $\Gamma \triangleright M\colon A$, the term M is SN. This applies to kinds, types, and terms (proofs).*

An interesting consequence of theorem E.2 is an elementary proof of the consistency of CC. There are other elementary methods for showing that CC is consistent, for example, the "proof-irrelevance semantics", which consists in interpreting types as Zermelo-Fraenkel sets, and $\star$ as the set $\{0, 1\}$ (for details, see Coquand [11]). What is more interesting, is that theorem E.2 can be used to show in an elementary fashion that certain contexts are consistent, as shown in Coquand [11].

DEFINITION E.16 We say that a context Δ is *consistent* iff there is some valid type σ (with $\Delta \triangleright \sigma\colon\star$) such that $\Delta \triangleright M\colon\sigma$ is **not** provable for any (proof) term M. We also say that a type σ is *inhabited* in the context Δ iff there is some (proof) term M such that $\Delta \triangleright M\colon\sigma$ is derivable.

Saying that CC is consistent means that the empty context is consistent, which is equivalent to the fact that some valid closed type is *not* inhabited. An elegant combinatorial proof of the consistency of CC using theorem E.2 is given below.

LEMMA E.17 *The theory CC is consistent. Furthermore, the valid type $(\Pi x{:}\star)x$ is not inhabited.*

Proof. First, observe that the judgment $x{:}\star \triangleright x{:}\star$ is derivable, and so $\triangleright\ (\Pi x{:}\star)x{:}\star$ is derivable. We make use of the following crucial fact: If M is a valid proof in some context Δ and M is a normal form w.r.t. β-reduction, then M is of the shape

$$\lambda x_1{:}\,A_1.\ \ldots.\ \lambda x_m{:}\,A_m.\, yN_1 \ldots N_n,$$

where y is a variable possibly among $x_1, \ldots, x_m$, and $N_1, \ldots, N_n$ are normal forms $(m, n \geq 0)$, but not necessarily of the same shape as M, since some N_i's could be products.

The above fact is easily shown by induction on the size of M. The case where $M = M_1M_2$ is the only interesting one. Because M is normal, M_1 cannot be an abstraction. However, it must be a proof, and by the induction hypothesis, it must be either a variable or an application of the form $xN_1 \ldots N_n$.

Now, assume that there is a valid closed proof M such that $\triangleright\ M{:}\,(\Pi x{:}\star)x$ is derivable. By theorem E.2 and by lemma D.11, we can assume that M is in normal form. But then, it is easily seen that it must be the case that we have $M = \lambda x{:}\star.\, yN_1 \ldots N_n$ and that we have a derivation $x{:}\star \triangleright yN_1 \ldots N_n{:}\,x$. However, it is a simple property of CC that for every judgment $\Delta \triangleright E$, $FV(E) \subseteq dom(\Delta)$. This implies that $y = x$. However, x is now both a proof and a type, which is impossible by lemma D.12. $\square$

In Coquand [11], it is shown using theorem E.2 that a nontrivial context *Inf* is consistent. The proof is elementary, except for the use of theorem E.2. As we shall see below, there cannot be any elementary direct proof of the consistency of the context *Inf* (say in first-order Peano arithmetic, or even in classical higher-order arithmetic). Letting $Void = (\Pi x{:}\star)x$ (the "absurd" type),

$$\begin{aligned} Inf = A{:}\star,\ & f{:}\,A \to A,\ R{:}\,A \to A \to \star, \\ & h_1{:}\,(\forall x{:}\,A)(Rxx \to Void), \\ & h_2{:}\,(\forall x, y, z{:}\,A)(Rxy \to Ryz \to Rxz), \\ & h_3{:}\,(\forall x{:}\,A)Rx(fx). \end{aligned}$$

The context *Inf* can be viewed as a kind of axiom of infinity. In turn, it can be shown that the consistency of this context implies the consistency of classical higher-order arithmetic. The proof is elementary, except for the use of theorem E.2. Thus, by Gödel's second incompleteness theorem, we obtain that strong normalization in CC (theorem E.2) is **not** provable in classical higher-order arithmetic.

Theorem E.2 and the Church-Rosser property also imply the decidability of type-checking in CC. In fact, a stronger result holds. The main lines of a proof of the above result were given by the first author in a communication to the "Types forum". This proof is quite similar to a proof by Martin Löf [6].

LEMMA E.18 *Given any context $\Delta = x_1: A_1, \ldots, x_n: A_n$ and any expression M, it is decidable whether $\Delta \triangleright$, and if so, whether $\Delta \triangleright M: kind$ or $\Delta \triangleright M: A$ for some A (which is given by the algorithm).*

Proof sketch. There are two kinds of problems: testing whether $\Delta \triangleright$ or $\Delta \triangleright \star: kind$, and testing whether M kind/type-checks in the context Δ. We associate a complexity measure to these two problems as follows. Let $c(\langle x_1: A_1, \ldots, x_n: A_n \rangle) = 1+$ the sum of the sizes of each A_i (and the same value for $c(\langle x_1: A_1, \ldots, x_n: A_n \rangle, \star))$, and $c(\langle x_1: A_1, \ldots, x_n: A_n \rangle, M) =$ the size of $M+$ the sum of the sizes of each A_i. We proceed by induction on complexity measures. There are several cases.

1. The problem is $\Delta \triangleright ?$ or $\Delta \triangleright \star: kind?$ and $\Delta = \emptyset$. The answer is yes.

2. The problem is $x_1: A_1, \ldots, x_n: A_n \triangleright$ or $x_1: A_1, \ldots, x_n: A_n \triangleright \star: kind$ and $n > 0$. Check whether A_n is well formed in $x_1: A_1, \ldots, x_{n-1}: A_{n-1}$. If the algorithm returns B, check that either the normal form of B is $\star$, or that B is a kind.

3. M is a variable x. Check whether A_n is well formed in $x_1: A_1, \ldots, x_{n-1}: A_{n-1}$. If the algorithm returns B, check that the normal form of B is $\star$ or that B is a kind, and whether x is one of the x_i.

4. M is of the form $(\Pi x: A)B$. Check whether A is well formed in $x_1: A_1, \ldots, x_n: A_n$ and whether B is well formed in $x_1: A_1, \ldots, x_n: A_n, x: A$.

5. M is of the form $\lambda x: A.\, N$. Check whether A is well formed in $x_1: A_1, \ldots, x_n: A_n$ and whether N is well formed in $x_1: A_1, \ldots, x_n: A_n, x: A$. If the answer to the second problem is yes and the algorithm returns P, then $x_1: A_1, \ldots, x_n: A_n \triangleright M: (\Pi x: A)P$.

6. M is of the form $M_1 M_2$. This case requires the fact that every term has a unique normal form. First, we check whether both M_1 and M_2 are well-formed in $x_1: A_1, \ldots, x_n: A_n$. If so, we check whether the normal form of the type/kind of M_1 is of the form $(\Pi x: A)P$ and the normal form of the type/kind of M_2 is P. $\square$

A closer look at definition E.5, especially the definitions of $[\![\Gamma \triangleright (\Pi x: K)D]\!]\rho\Delta$ and $[\![\Gamma \triangleright (\Pi x: \sigma)D]\!]\rho\Delta$, suggests the definition of certain dependent products. Let Δ be a context and $(\Pi x: K)D$ be a kind or a type such that $\Delta \triangleright (\Pi x: K)D: \kappa$, $\kappa \in \{\star, kind\}$, with K a kind.

DEFINITION E.19 Let $\mathcal{A} = (\mathcal{A}_{\Delta'})_{\Delta' \supseteq \Delta}$ be any Δ'-indexed family of candidates such that $\mathcal{A}_{\Delta'} \in \mathcal{C}_{K,\Delta'}$, and let F be any function with domain $\{\langle \Delta' \triangleright A, C \rangle \mid \Delta' \triangleright A: K,\ \Delta' \supseteq \Delta, \text{ and } C \in \mathcal{C}_{A,\Delta'}\}$, and such that $F(\Delta' \triangleright A, C) \in \mathcal{C}_{D[A/x],\Delta'}$. The dependent product $\prod(\mathcal{A}, F; (\Pi x: K)D)$ is defined as follows:

$$\begin{aligned}\prod(\mathcal{A}, F; (\Pi x: K)D) = \{\Delta' \triangleright M \mid\ & \Delta' \triangleright M: (\Pi x: K)D,\ \Delta' \supseteq \Delta, \text{ and} \\ & \forall \Delta'' \supseteq \Delta',\ \forall \Delta'' \triangleright A \in \mathcal{A}_{\Delta''},\ \forall C \in \mathcal{C}_{A,\Delta''}, \\ & \Delta'' \triangleright (MA) \in F(\Delta'' \triangleright A, C)\}.\end{aligned}$$

Let Δ be a context and $(\Pi x: \sigma)D$ be a kind or a type such that $\Delta \triangleright (\Pi x: \sigma)D: \kappa$, $\kappa \in \{\star, kind\}$, with σ a type.

Definition E.20 Let $\mathcal{A} = (\mathcal{A}_{\Delta'})_{\Delta' \supseteq \Delta}$ be any Δ'-indexed family of candidates such that $\mathcal{A}_{\Delta'} \in \mathcal{C}_{\sigma,\Delta'}$, and let F be any function with domain $\{\Delta' \triangleright N \mid \Delta' \triangleright N{:}\, \sigma,\ \Delta' \supseteq \Delta\}$, and such that $F(\Delta' \triangleright N) \in \mathcal{C}_{D[N/x],\Delta'}$. The dependent product $\prod(\mathcal{A}, F; (\Pi x{:}\,\sigma)D)$ is defined as follows:

$$\begin{aligned}\prod(\mathcal{A}, F; (\Pi x{:}\,\sigma)D) = \{\Delta' \triangleright M \mid\ & \Delta' \triangleright M{:}\,(\Pi x{:}\,\sigma)D,\ \Delta' \supseteq \Delta, \text{ and} \\ & \forall \Delta'' \supseteq \Delta',\ \forall \Delta'' \triangleright N \in \mathcal{A}_{\Delta''}, \\ & \Delta'' \triangleright (MN) \in F(\Delta'' \triangleright N)\}.\end{aligned}$$

Then, we can express $[\![\Gamma \triangleright (\Pi x{:}\,K)D]\!]\rho\Delta$ and $[\![\Gamma \triangleright (\Pi x{:}\,\sigma)D]\!]\rho\Delta$ as dependent products:

$$[\![\Gamma \triangleright (\Pi x{:}\,K)D]\!]\rho\Delta = \prod(([\![\Gamma \triangleright K]\!]\rho\Delta')_{\Delta' \supseteq \Delta},\ F;\ ((\Pi x{:}\,K)D)[\rho]),$$

where F is the function such that

$$\langle \Delta' \triangleright A, C\rangle \mapsto [\![\Gamma, x{:}\,K \triangleright D]\!]\rho[x{:=}\,\langle A, C\rangle]\Delta',$$

with $\Delta' \triangleright A{:}\,K[\rho]$ and $C \in \mathcal{C}_{A,\Delta'}$, and

$$[\![\Gamma \triangleright (\Pi x{:}\,\sigma)D]\!]\rho\Delta = \prod(([\![\Gamma \triangleright \sigma]\!]\rho\Delta')_{\Delta' \supseteq \Delta},\ F;\ ((\Pi x{:}\,\sigma)D)[\rho]),$$

where F is the function such that

$$\Delta' \triangleright N \mapsto [\![\Gamma, x{:}\,\sigma \triangleright D]\!]\rho[x{:=}\,N]\Delta',$$

with $\Delta' \triangleright N{:}\,\sigma[\rho]$.

The definition of $\prod(\mathcal{A}, F; (\Pi x{:}\,\sigma)D)$ is inspired by the definition of the dependent product $\prod(A, F)$ given by Coquand and Huet on page 107 of their paper [4]. The difference is that Coquand and Huet give a definition of $\prod(A, F)$ for *untyped* λ-terms. They have no definition analogous to our dependent product $\prod(\mathcal{A}, F; (\Pi x{:}\,K)D)$ where K is a kind. Also, Coquand and Huet's main theorem on page 109 of their paper [4], can be considered as a version of our theorem E.1 for "stripped terms" (that is, valid terms of CC from which type information has been erased). However, theorem E.1 is a stronger result, since it yields theorem E.2 as a corollary, whereas Coquand and Huet's theorem only shows that the *type erasure* $Erase(M)$ of any valid term of CC is SN. As far as we know, there does not seem to be any way to infer from the fact that $Erase(M)$ is SN that M itself must be SN. This is in contrast with the situation in $\lambda^{\forall}$ (and system F_{ω}).

We now examine the special case of LF, and note that strong normalization holds as a corollary, but does not make any use of families of candidates. Only the canonical $can_{A,\Delta}$ are needed.

F Strong Normalization in LF

Since LF can be viewed as a fragment of CC obtained by disallowing products and abstractions over type variables, it follows immediately from theorem E.2 that all valid terms of LF are strongly normalizing (under β-reduction). However, it turns out that the powerful artillery of the $\mathcal{C}_{A,\Delta}$ is unnecessary to prove this result. In *LF*, we can only have products of the form $(\Pi x{:}\sigma)D$, and abstractions of the form $\lambda x{:}\sigma.\, B$, when σ is a *type* (but *not a kind*). Thus, we have a simpler definition of $[\![\Gamma \triangleright A]\!]\rho\Delta$. Again, A is either a type family or a kind valid in Γ, and the definition only makes sense when ρ satisfies Γ at Δ.

DEFINITION F.1 In the clauses below, K stands for a kind, σ for a type, A, B for type families, D for a kind or a type, M for a type family or a term (proof), and N for a term (proof).

$$\begin{aligned}
[\![\Gamma \triangleright kind]\!]\rho\Delta &= \rho_c(kind),\\
[\![\Gamma \triangleright \star]\!]\rho\Delta &= \rho_c(\star),\\
[\![\Gamma \triangleright x]\!]\rho\Delta &= \rho_c(x),\\
[\![\Gamma \triangleright AB]\!]\rho\Delta &= [\![\Gamma \triangleright A]\!]\rho\Delta(\Delta \triangleright B[\rho],\ [\![\Gamma \triangleright B]\!]\rho\Delta),\\
[\![\Gamma \triangleright AN]\!]\rho\Delta &= [\![\Gamma \triangleright A]\!]\rho\Delta(\Delta \triangleright N[\rho]),\\
[\![\Gamma \triangleright (\Pi x{:}\sigma)D]\!]\rho\Delta &= \{\Delta' \triangleright M \mid \Delta' \triangleright M{:}((\Pi x{:}\sigma)D)[\rho],\ \Delta' \supseteq \Delta,\ \text{and}\\
&\qquad \forall\Delta'' \supseteq \Delta',\ \forall\Delta'' \triangleright N \in [\![\Gamma \triangleright \sigma]\!]\rho\Delta'',\\
&\qquad \Delta'' \triangleright (MN) \in [\![\Gamma, x{:}\sigma \triangleright D]\!]\rho[x{:}= N]\Delta''\},\\
[\![\Gamma \triangleright \lambda x{:}\sigma.\, B]\!]\rho\Delta &= \lambda(\Delta' \triangleright N).\, [\![\Gamma, x{:}\sigma \triangleright B]\!]\rho[x{:}= N]\Delta',\\
&\qquad \text{a function with domain}\\
&\qquad \{\Delta' \triangleright N \mid \Delta' \triangleright N{:}\sigma[\rho],\ \Delta' \supseteq \Delta\}.
\end{aligned}$$

Remarkably, the candidates, that is, the sets $C \in \mathcal{C}_{A,\Delta}$, *do not* appear anywhere in these definitions. The only place where they play a role is in $[\![\Gamma \triangleright x]\!]\rho\Delta$ and $[\![\Gamma \triangleright \star]\!]\rho\Delta$. However, this role is very passive. In fact, all we need to establish strong normalization is to assign the canonical sets and functions $can_{A,\Delta}$. More precisely, $\rho_c(kind)$ is the set $can_{\star,\Delta}$ of SN kinds, $\rho_c(\star)$ is the set $can_{\star,\Delta}$ of SN types, and $\rho_c(x) = can_{\Gamma(x)[\rho],\Delta}$. Only the substitution component ρ_s of ρ needs to be arbitrary for the proof to go through, the other component ρ_c remaining constant (and determined by the canonical elements). Thus, the proof of strong normalization for *LF* uses little more than is needed for the proof of strong normalization in the simply-typed λ-calculus, namely the existence of the canonical sets and functions, which itself depends on the existence of the measure $c(K)$, where K a kind. This is not surprising in view of another proof by Harper, Honsell, and Plotkin [14], in which a mapping from *LF* into the simply-typed λ-calculus is used. It should be noted that their proof applies to β and η reduction, but we do not know presently how to extend our approach to η-reduction.

G Other Proofs

This section lists other proofs of normalization or strong normalization that we are aware of, in chronological order. We apologize if we are unaware of other proofs not mentioned here. To us, the history of this proof seems sufficiently interesting to be told, especially in a preliminary report, even if it is incomplete. It has been reported that some of these proofs contain errors. We are indeed aware of some errors, and we will briefly mention what they are. We apologize for any (unintentional) omissions or misinterpretations.

1. Coquand, January 1985 [2]. This is Thierry Coquand's thesis. A proof of normalization is given, a well as some indications on how to extend it to strong normalization. There is a problem with the definition of the sets $\mathcal{C}_{A,\Delta}$ when A is a type family of kind $(\Pi x{:}\,B)D$. The members of $\mathcal{C}_{A,\Delta}$ are indeed functions, but only of one argument, the candidate argument. A similar problem arises in the definition of $[\![\Gamma \triangleright \lambda x{:}\,K.\,B]\!]\rho\Delta$, where the argument $\Delta' \triangleright A$ is omitted. As a consequence, $[\![\Gamma \triangleright A]\!]\rho\Delta$ is not always well-defined.

2. Jutting, December 1986 [7]. This is a note attempting to correct Coquand's proof of normalization given in his thesis. The introduction mentions discussions with Coquand, leading to this note. As we see it, the definition of $[\![\Gamma \triangleright A]\!]\rho\Delta$ is indeed repaired correctly. However, Δ is dropped from the $\mathcal{C}_{A,\Delta}$, which becomes a family of sets of *closed* terms. To insure that each $C \in \mathcal{C}_A$ is nonempty (A a closed type family or closed kind), Jutting adds a countably infinite set of constants. Unfortunately, this causes a problem. Indeed, the language has now been enriched, new types can be formed, and some new closed types may not be inhabited.

3. Coquand, 1987 [3]. This is a note in which Coquand fixes the problem with the addition of new constants, and gives a proof of strong normalization for the first time. The proof uses infinite contexts, and basically Henkin's technique for adding new witnesses, so that all closed types are inhabited.

4. Pottinger, February 1987 [8]. This paper refers to Coquand 1987, and gives a proof of strong normalization apparently inspired by Coquand's proof. Infinite contexts are also used, as well as an idea due to Seldin. Although we need to examine it more closely, the proof seems correct, but rather difficult to follow.

5. Seldin, November 1987 [10]. This is a report, "Mathesis: the Mathematical Foundations of Ulysses", in which a proof of strong normalization for a *variant* of the theory of constructions is given. We have not yet had the time to examine this proof carefully, but it appears that it also uses infinite contexts. It appears to be more along the line of Martin Löf's proof of normalization for F_ω, defined as a Prawitz-style natural deduction system.

6. Zhaohui Luo, 1989 [15]. There is apparently a proof of strong normalization for an extension of CC with universes, given in Luo's thesis. We do not have this document yet.

7. Geuvers and Nederhof, June 1989 [9]. The authors present what they call a modular proof of strong normalization, by reducing strong normalization in CC to strong normalization in Girard's F_ω. This is accomplished by defining a mapping from CC to F_ω, such that reduction of terms is preserved. Strong normalization for the terms of F_ω is itself reduced to strong normalization for the erased (raw) terms of F_ω, which is proved directly.

8. Berardi, 1989 [16]. Berardi gives a proof (apparently due to Terlouw) in an appendix of his thesis.

Acknowledgment: We wish to thank Val Breazu-Tannen and Sunil Shende for many helpful comments.

References

[1] Mitchell, J. C. and Moggi, E., Kripke-style models for typed lambda calculus, Second Symposium on Logic in Computer Science, June 22-25 1987, pp 303-314.

[2] Coquand, Thierry, Une Théorie Des Constructions. Thèse de 3^{eme} Cycle, Université Paris VII, January 1985.,

[3] Coquand, Thierry, Metamathematical Investigations of a Calculus of Constructions. Privately circulated manuscript, 1987. In Rapport technique INRIA 110, 1989.

[4] Coquand, Thierry and Huet, G., The Calculus of Constructions. Information and Computation, 76, 2/3, 1988, pp 95-120.

[5] Ranta, A., Constructing Possible Worlds. To appear, Theoria, 1990.

[6] Martin Löf, P., An Intuitionistic Theory of Types. Privately circulated manuscript, University of Stokholm, 1972,

[7] van Benthem Jutting, L. S., Normalization in Coquand's System, Private Communication, December 1986.

[8] Pottinger, G., Strong Normalization for Terms of the Theory of Constructions, Odyssey Research Associates, Feb. 1987.

[9] Geuvers, H. and Neherhof, M.-J., A Modular Proof of Strong Normalization for the Calculus of Constructions. Submitted for publication, Journal of Functional Programming.

[10] Seldin, J., Mathesis: The Mathematical Foundations of Ulysses. Technical Report RADC-TR-87-223, Odyssey Research Associates, 1987.

[11] Coquand, Thierry, Metamathematical Investigations of a Calculus of Constructions. In Logic And Computer Science, P. Odifreddi, ed, Academic Press, 1990, pp 91-122.

[12] Gallier, J., On Girard's "Candidats de Reductibilités", In Logic And Computer Science, P. Odifreddi, ed, Academic Press, 1990, pp 123-203.

[13] Girard, Jean Yves, Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Thèse de Doctorat d'Etat, Université Paris VII, 1972.

[14] R. Harper and F. Honsell and G. Plotkin, A Framework for Defining Logics. Submitted for publication, J. ACM.

[15] Luo, Z., ECC, an extended calculus of constructions. Forthcoming thesis, University of Edinburgh.

[16] Berardi, S., ? Universita di Torino, 1989.