PROCEEDINGS OF THE

SECOND WORKSHOP ON

LOGICAL FRAMEWORKS

Edinburgh

May 1991

Eds G. Huet, G. Plotkin & C. Jones

Contents

⋆Logical Frameworks⋆

Foreword

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

This workshop was held at St Cecilia’s Hall in Edinburgh from the 19th to the 24th of May 1991; it was attended by 93 people. The programme was put together by Gerard Huet and Gordon Plotkin. Local arrangements were looked after by George Cleland, Monica Lekuse and Gordon Plotkin. Paul Anderson and George Cleland took care of computing provision.

These preliminary proceedings have been collected from LaTeX sources, using electronic mail and spliced together (thanks to all who read the guidelines). They comprise 19 of the 28 papers presented at the meeting.

This document may be obtained by anonymous ftp from Edinburgh University (compressed file `export/bra/proc91.dvi.Z` on “cumbrae”, internet address 129.215.160.150, login name “anonymous”).

Workshop Programme: Logical Frameworks, 1991

Sunday, May 19th

6.30–9.30pm	Reception
7.30–8.30pm	Concert Strathclyde University Chamber Choir

Monday, May 20th

9.00–9.10am	Welcome to Workshop
9.10–10.00am	David Basin and Robert Constable Invited Talk: Meta-logical frameworks
10.00–10.30am	Tobias Nipkow Order-Sorted Polymorphism
10.30–11.00am	Coffee Break
11.00–11.45am	Christine Paulin-Mohring Properties of Programs extracted from proofs in the Calculus of Inductive Definitions
11.45–12.30pm	Yves Bertot An Effective Calculus of Residuals
12.30–2.00pm	Lunch
2.00–3.00pm	Presentation of Demonstrations **Goteborg Demo** Thierry Coquand, Bengt Nordstrom, Lennart Augustsson: ALF **Cambridge Demo** Tom Melham: Theorem-proving tools in the HOL system. **Edinburgh Demo** Randy Pollack: LEGO **INRIA Rocquencourt Demo** Gerard Huet, Amy Felty, B. Werner, H. Herbelin, Gilles Dowek: Presenting the system Coq, version 5.6.
3.00–4.30pm	Demonstration sessions, (tea available)
4.30–5.00pm	Discussion of Demonstrations

Tuesday, May 21st

9.00–10.00am	Tom Melham Invited Talk: A Mechanized Theory of the pi-calculus in HOL
10.00–10.30am	Claire Jones Completing the Rationals and Metric Spaces in LEGO
10.30–11.00am	Coffee Break
11.00–11.45am	Nora Szasz A Machine checked proof that Ackermann's function is not primitive recursive
11.45–12.30pm	Per Martin-Lof Choice sequences in type theory
12.30–2.00pm	Lunch
2.00–3.00pm	Presentation of Demonstrations **Cambridge Demo** Larry Paulson, Toby Nipkow: Polymorphic Isabelle. **Edinburgh Demo** Alan Smaill, Colin Phillips: The CLAM proof planner. **INRIA Sophia-Antipolis Demo** Laurent Thery: Using Centaur as a front end for theorem provers Paris VII Demo M. Parigot, P. Manoury, M. Simonot, C. Raffalli: PROPRE: an experimental language for programming with proofs.
3.00–4.30pm	Demonstration sessions, (tea available)
4.30–5.00pm	Discussion of Demonstrations

Wednesday, May 22nd

9.00–10.00am	N.G. de Bruijn Invited Talk: Algorithmic correctness check for the delta-lambda calculus
10.00–10.30am	Yves Lafont Programming languages inspired by classical logic
10.30–11.00am	Coffee Break
11.00–11.45am	F Barbanera and Stefano Berardi Program Extraction for Classical logic
11.45–12.30pm	Michel Parigot Free Deduction: an analysis of computations in classical logic
12.30–2.00pm	Lunch
2.00–5.30pm	EC Review (details to be confirmed)

Thursday, May 23rd

9.00–10.00am	Dale Miller Invited Talk: Unification of Simply Typed Lambda-Terms as Logic Programming
10.00–10.30am	David Pym Towards an Algebraic Framework for Defining Sequential Logics
10.30–11.00am	Coffee Break
11.00–11.45am	Razvan Diaconescu and Joseph Goguen Logical Support for Modularisation
11.45–12.30pm	Thierry Coquand A question about classical logic
12.30–2.00pm	Lunch
2.00–2.45pm	Herman Geuvers Iteration and Recursion on Inductive and Coinductive Types
2.45–3.30pm	Jan Smith An interpretation of Kleene's slash in type theory
3.30–4.00pm	Tea
4.00–4.45pm	Gilles Dowek Proof synthesis in Barendregt's cube
4.45–5.30pm	Gilles Amiot Second order unification and some applications
6.30pm	Leave for Banquet at Dalhousie Castle Hotel

Friday, May 24th

9.00–10.00am	Robert Harper Invited Talk: A Calculus of Modules for LF and Elf
10.00–10.30am	Amy Felty A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs
10.30–11.00am	Coffee Break
11.00–11.45am	Philippa Gardner ELF+: a type theory for representing logics
11.45–12.30pm	Sean Matthews, David Basin and Alan Smaill Experience with FS0 as a Framework Theory
12.30–2.00pm	Lunch
2.00–2.45pm	Z Luo Data Refinement in Type Theory
2.45–3.30pm	Chet Murthy Extracting proofs from Classical Logic
3.30–4.00pm	Tea
4.00–4.45pm	Vincent Danos New ways to look at lambda-calculus: pure nets and dynamic algebras
4.45–5.30pm	Stefano Berardi Program extraction for Predicative Systems.

Abstracts of Demonstrations

Cambridge Demo

Tom Melham

Theorem-proving tools in the HOL system

Abstract

The HOL system is an LCF-style proof assistant for reasoning in a version of higher order logic based on Church's formulation of simple type theory. The LCF approach to theorem proving is fairly well known, so this demonstration of the HOL system will concentrate primarily on the theorem-proving tools that have been built on top of the system, rather than on the core system itself.

Cambridge Demo

Larry Paulson, Toby Nipkow

Polymorphic Isabelle

Abstract

Isabelle has been extended with order-sorted polymorphism. The sort system for type variables allows object-logics to exploit the meta-level polymorphic type system in a controlled fashion. Many-sorted first-order logic is obtained by declaring polymorphic quantifiers, where the type variables may range over, say, the types of integers and lists (for suitable element types). Higher-order logic is obtained by allowing the type variables to range over function types and the type of truth values.

The demonstration will present the formalization of higher-order logic and demonstrate proofs involving a formalization of propositional logic within higher-order logic. This illustrates the use of Isabelle to reason inductively about the structure of proofs.

Edinburgh Demo

Randy Pollack

LEGO

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 SN of second-order lambda calculus (Berardi), Tarski's fixpoint theorem, the binomial theorem for rings, some elementary topology and metric spaces, 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, lack of "inductive types", and the notion of tactics. Currently we are experimenting with adding reductions to a context (as in ALF). 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.

Edinburgh Demo

Alan Smaill, Colin Phillips

The CLAM proof planner

Abstract

The CLAM system takes a meta-level description of a theorem-proving task, and a number of tactic specifications (known as methods), in order to return a combination of tactics that are likely to yield a proof of a given conjecture. We show the system in use in the domain of inductive proofs, interfaced to different implementations of type theory.

Goteborg Demo

Thierry Coquand, Bengt Nordstrom, Lennart Augustsson

ALF

Abstract

ALF (Another Logical Framework) is an implementation of a logical framework. It runs under X-windows and uses windows extensively to make derivations easier.

INRIA Rocquencourt Demo

Gerard Huet, Amy Felty, B. Werner, H. Herbelin Gilles Dowek

Presenting the system Coq, version 5.6.

Abstract

Coq is an implementation of the Calculus of Inductive Constructions, under development at INRIA Rocquencourt and ENS Lyon. We first show Gilbreath's trick (G. Huet), which illustrates the methodology for proving inductive mathematical statements in a top-down fashion, using a basic set of 5 tactics. Next, we present our new user interface, using the CAML widgets of D. de Rauglaudre (A. Felty). We then present the extraction of running ML programs from inductive proofs (B. Werner). As an application, we demonstrate the behaviour of a program corresponding to an A-translation of a classical proof of Higman's lemma (H. Herbelin). Finally, we present a prototype implementation of a new engine with existential variables for proof synthesis (G. Dowek).

INRIA Sophia-Antipolis Demo

Laurent Thery

Using Centaur as a front end for theorem provers

Abstract

The demonstration comprises two parts:

- first, we use Amy Felty's theorem prover, build a proof in the Intuitionistic Sequent Calculus, and display it in different formats: Sequent proof, Natural Deduction Proof tree, and textual proof. The prover is written in lambda-Prolog, and we use the Z implementation of lambda-Prolog, which is 15 times faster than elp.
- second, we use HOL as a logical engine; the front end is more sophisticated due to the complexity of the HOL environment.

This is ongoing work of Laurent Théry for his Ph.D. thesis.

Paris VII Demo

M. Parigot, P. Manoury, M. Simonot, C. Raffalli

PROPRE: an experimental language for programming with proofs

Abstract

PROPRE is an experimental programming language based on the paradigm "proofs as programs". Programs are terms of λ-calculus extracted from proofs of termination of functions defined by sets of equations, using deterministic proof strategies. Each strategy is based on a particular form of induction. In this demo we will show examples of derivation of programs using these strategies. We will show what we are able to do and what we are not able to do, at the present time.

Abstracts of Talks

Gilles Amiot

Second order unification and some applications

Abstract

Second order unification is an extension of standard unification (defined by Herbrand, Robinson). This extension is obtained by adding to the language second order variables. There are two possible extensions: the first is to add function variables, and the second is to add predicate variables. We present the results on the decision problem for both cases (the first was completely investigated by Goldfarb and Farmer, and the second by Amiot). Unification has numerous applications. In particular, it has applications in proof theory and proof search. In order to illustrate these applications, we show how decision problems in linear logic can be linked with unification. In particular we derive new undecidability results for linear logic from undecidability results for second order unification.

F Barbanera and Stefano Berardi

Program Extraction for Classical logic

Abstract

Statman and Prawitz considered Intuitionistic Natural Deduction System for logic and arithmetic, with the non-logical part reduced to (maybe induction and) atomic rules of the form:

$$\frac{A_1, \ldots, A_n}{A}$$

(with $A_1, \ldots, A_n, A$ atomic). Then they added, together with double negation rule, reductions to eliminate the use of double negation for maximum formulas (Statman), or for non-atomic formulas (Prawitz). For arithmetical systems the double negation rule for atomic formulas is constructively correct . A sample of Statman reductions is:

$$\cfrac{\cfrac{\begin{matrix}[\neg A \& B] \\ \vdots \\ \mathit{False}\end{matrix}}{A \& B}}{A}$$

reduces to:

$$\cfrac{\cfrac{[\neg A] \qquad \cfrac{[A\&B]^1}{A}}{False}\ 1}{\cfrac{[\neg A\&B]}{\begin{array}{c}\vdots\\ \cfrac{False}{A}\ \text{Double Neg.}\end{array}}}$$

One cannot define such reductions for $\vee$ and $\exists$; thus, one is forced to put:

$$\begin{aligned} A \vee B &= \neg(\neg A \& \neg B) \\ \exists x A &= \neg(\forall x \neg A) \end{aligned}$$

One obtains, that way, a Classical Natural Deduction System for Arithmetic. The main technical results for it are:

1. Strong Normalization and Church-Rosser;
2. A closed normal proof of False contains only closed atomic formulas, or negations of closed atomic formulas.

From 1 and 2 one may infer consistency, as did Prawitz. In our paper, we use it, instead, to extract the "witness" of an existential statement, in the following way:

If one has a **closed** classical proof of $(\exists x(f(x) = 0))$, just add the **atomic** rule:

$$\frac{f(x) = 0}{False}$$

and get a **closed** proof of *False* (of course, the new rule is semantically incorrect!).

Points 1 and 2 hold in the extended system, because we only added an atomic rule. By 1, one obtains a closed normal proof of *False* ; by 2, such a proof contains only closed atomic formulas, or their negations. Thus, one proves *False* using only a **finite** number of times this new rule, say:

$$\frac{f(a_1) = 0, \ldots, f(a_n) = 0}{False}$$

($a_1, \ldots, a_n$ closed values of type Integer) and this forces $f(a_1) = 0$, or $\ldots$, or $f(a_n) = 0$.

Of course, the problem is to pick up some a_i such that $f(a_i) = 0$; some non-deterministic choice must be allowed. Prawitz's suggestion was to add the following new reduction: replace a proof of an atomic A with any proper subproof of the same A (if any!), **proviso** that the assumptions of the subproofs are included in the assumptions of the proof. With such a new reduction, we prove that we get rid of all applications of the new rule, but one at the end:

$$\frac{f(a_i) = 0}{False}$$

This forces $f(a_i) = 0$ to be true, as wished. We use a system exploiting ideas similar to Kripke forcing semantics to explain and obtain the technical results outlined above.

David Basin Robert Constable

Meta-logical frameworks

Abstract

In computer science we speak of *implementing* a logic; this is done in a programming language, such as Lisp, called here the *implementation language*. We also reason about the logic, as in understanding how to search for proofs or in knowing it is consistent; in the terminology of mathematical logic, these arguments are conducted in the *metalanguage* of the *object language* being implemented. We also reason about the implementation itself, say to know it is correct. This is done in a *programming logic*. How do all these logics relate? This paper considers that question and more.

We show that by taking the view that the metalogic is primary, these other ones are related to it in standard ways. The metalogic must be suitably rich so that the object logic can be presented as an abstract data type, and it must be suitably computational (or constructive) so that an instance of the type is an implementation. The data type abstractly encodes all that is relevant for metareasoning, i.e. not only the term constructing functions but also the principles for reasoning about arbitrary terms and computing with them.

Our work can also be seen as an approach to the task of finding a generic way to present logics and their implementations, which is for example the goal of the Edinburgh Logical Frameworks (LF) effort. This approach extends well beyond proof-construction and includes computational metatheory as well.

Stefano Berardi

Program extraction for Predicative Systems

Abstract

We consider two extensions of an algorithm, due to Takayama, which simplifies proofs of a specification to programs which satisfy the given specification. We also prove, using a naive set-theoretical semantic, that the simplified algorithms have still the same input/output behaviour of the original ones, and thus that they satisfy the original specification.

Yves Bertot

An Effective Calculus of Residuals

Abstract

Considering a natural semantics specification of lambda-calculus, we show that a formal manipulation of the inference rules permits to introduce effective residual computations. These computations have applications in all kinds of marked reduction systems used in proving theorems in the lambda calculus, e.g. the finite developments theorem etc.

Thierry Coquand

A question about classical logic

Abstract

We present Gentzen's first proof of consistency of arithmetic formulated in the framework of inductive definitions. This reformulation suggests naturally a semantics of evidence for classical arithmetic. While toying with an example of Gabriel Stolzenberg that involves a typically classical lemma (infinite box principle), it appeared that there is another way of doing the cut-elimination than Gentzen's method. We present this algorithm, its proof of partial correctness, and the conjecture of total correctness.

Vincent Danos

New ways to look at lambda-calculus: pure nets and dynamic algebras

Abstract

Pure nets and dynamic algebra are two new models of computation directly issued from linear logic. The talk will give an elementary presentation of these models, state their basic computational properties and compare them to lambda-calculus.

N.G. de Bruijn

Algorithmic correctness check for the delta-lambda calculus

Abstract

The delta lambda calculus is a lambda-typed lambda calculus with facilities for built-in definitions. A term in that calculus can efficiently represent a complete mathematical theory, with axioms, definitions, theorems and proofs. Correctness of a theory can therefore be interpreted as the correctness of a single lambda term. The talk will describe an algorithm that provides the essential kernel for a correctness checking program. Given a lambda term, that kernel produces, more or less in linear time, a list of subtasks. To every application occurring in the term there corresponds exactly one subtask. The subtasks are all of the following form: show that in the context C the type of A is definitionally equal to B.

Razvan Diaconescu and Joseph Goguen

Logical Support for Modularisation

Abstract

Modularisation is important for managing the complex structures that can arise in large theorem proving problems, and in large software and/or hardware development projects. This paper studies some properties of logical systems that support the definition, combination, parameterisation and reuse of modules. These properties include: (1) the preservation of various kinds of conservative extension under pushouts; (2) distributive laws for information hiding over sums; and (3) (Craig style) interpolation properties. Our results illuminate some interesting differences between equational style logics and first order style logics, encouraging us to conclude that, on the whole, equational style logics are more suitable for modularisation than first order style logics. We represent logical systems as institutions, and modules as sets of sentences. An important technical foundation is a novel categorical axiomatisation of the notion of "inclusion". We also show how to subsume the deduction-based approach of π-institutions under that of ordinary institutions, so that our results also apply to that setting.

Gilles Dowek

Proof synthesis in Barendregt's cube

Abstract

We sketch a few recent results in automated theorem proving in type systems of the Cube. We first present a complete proof synthesis method extending Huet's resolution. Because type systems of the cube verify the proofs-as-objects paradigm, the proof synthesis method is a one level process that merges unification and resolution. We also study two variants of this method, the first is incomplete but much more efficient and the second is a pre-unification algorithm. Then we present an algorithm for second order matching in all the systems of the cube and sketch proofs of undecidability of higher order matching in seven of these systems.

Amy Felty

A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs

Abstract

In this talk, we show that an intuitionistic logic with second-order function quantification, called hh^2, can serve as a meta-language to directly and naturally specify both sequent calculi and natural deduction inference systems for first-order logic. For the intuitionistic subset of first-order logic, we present a set of hh^2 formulas which simultaneously specifies both kinds of inference systems and provides a direct and concise account of the correspondence between cut-free sequential proofs and normal natural deduction proofs.

The logic of hh^2 can be implemented using such logic programming techniques as providing operational interpretations to the connectives and implementing unification on λ-terms. With respect to such an interpreter, our specification provides a description of how to convert a proof in one system to a proof in the other. The operation of converting a sequent proof to a natural deduction proof is functional in the sense that there is always one natural deduction proof corresponding to each sequent proof. Our specification, in fact, provides a direct implementation of the transformation in this direction.

Philippa Gardner

ELF+: a type theory for representing logics

Abstract

Using the framework ELF, we cannot distinguish which part of the type theory corresponds to the encoded logic without appealing to the particular encoding under consideration. We define a new framework, ELF+, based on ELF. With ELF+, we can identify the substructures of the type theory that correspond to the encoded logics. This enables us to characterize the notion of adequacy for such encodings.

When we represent a logic in ELF+, we identify the variables of the logic with certain variables of the type theory. Therefore, substitution for the object logics must behave in a similar way to substitution for the type theory. This motivates the presentation of the logics and their corresponding ELF+ substructures as indexed categories, so that encodings become functors between indexed categories. Adequate representation is then determined by conditions on such functors. Herman Geuvers Iteration and Recursion on Inductive and Coinductive Types. Abstract In his thesis T. Hagino introduces a typed lambda calculus with categorical data types, which includes both inductive and coinductive types. The reduction rules in this calculus only represent the iteration scheme and not recursion (which has to be "coded".) Coquand and Mohring encapsulate a notion of inductive type in a higher order type theory, which has real recursion and also captures the notion of inductive proof. In this talk we want to discuss the differences between iteration and recursion and extend a Coquand-Mohring-like higher order type system with coinductive types with a form of recursion.

Robert Harper

A Calculus of Modules for LF and Elf

Abstract

The logical framework LF is a language for defining logical systems based on the judgements-as-types principle. Elf, an implementation of LF, provides an operational interpretation of LF

types, so that LF signatures may be read as logic programs. As a result the LF lambda calculus is used both as a logic definition language and as a meta-programming language for manipulating object logics. In practice this methodology leads to relatively large signatures with a natural structure that cannot be adequately expressed in the basic LF lambda calculus. In particular, it would be desirable to support incremental construction of logic definitions and meta-programs (a form of "code inheritance"), parameterization and instantiation, and the interpretation of one system within another. In addition to the static structure of presentations, it is also important to provide mechanisms to initiate and control search. We present an initial design of a calculus of modules for LF and Elf, and illustrate its use with a number of examples.

[This talk represents joint work with Frank Pfenning.]

Claire Jones

Completing the Rationals and Metric Spaces in LEGO

Abstract

This talk will describe the work carried out in implementing/proving some elementary theorems of analysis in the Extended Calculus of Constructions using the LEGO system. Two main areas have been considered, firstly the completion of a metric space (defining a constructions function of type `Metric -> Metric`) and secondly the construction of the reals from the rationals. Many interesting issues arise from this e.g. how best to represent sets, subsets and functions.

Yves Lafont

Programming languages inspired by classical logic

Abstract

I will discuss the relations between:

1. the proof theory of intuitionistic and classical logic
2. exceptions and the continuations in functional programming.

This is inspired by the recent constructive interpretation of classical logic by J.Y. Girard.

Z Luo

Data Refinement in Type Theory

Abstract

We consider a type-theoretic approach to program specification and data refinement and show that a type theory with a strong logical power and good structural mechanisms provides a uniform and adequate formalism for modular development of programs and specifications.

Per Martin-Löf

Choice sequences in type theory

Sean Matthews, David Basin and Alan Smaill

Experience with FS_0 as a Framework Theory

Abstract

Feferman has proposed a logic, "FS_0", as an alternative framework for encoding logics and also for reasoning about those encodings. We have implemented a version of this system and performed experiments that support this claim. Specifically, we describe a formalization of predicate calculus and the development of a derived rule that manipulates formulas with bound variables. This application will be of interest to researchers working with frameworks which internalize bound variables in the meta-logic. We suggest that formalizing and reasoning about bound variables is not as difficult as is often supposed, and leads to a more powerful theory about the encoded logic.

Tom Melham

A Mechanized Theory of the pi-calculus in HOL

Abstract

The pi-calculus is a process algebra in the style of CCS, developed by Milner, Parrow and Walker for modelling concurrent systems in which the pattern of interconnection between processes can change over time. This talk will describe work in progress on a mechanized formal theory of the pi-calculus in higher order logic using the HOL theorem prover. The main aim of this work, which is being done jointly with Mike Gordon at the Computer Laboratory in Cambridge, is to provide a tool to support both reasoning about applications using the pi-calculus and metatheoretic reasoning about the calculus itself.

Dale Miller

Unification of Simply Typed λ-Terms as Logic Programming

Abstract

The unification of simply typed λ-terms modulo the rules of $\beta\eta$-conversion is often called "higher-order" unification because of the possible presence of variables of functional type. This kind of unification is undecidable in general and if unifiers exist, most general unifiers may not exist. In this paper, we show that such unification problems can be coded as a query of the logic programming language L_λ in a natural and clear fashion. In a sense, the translation only involves explicitly axiomatizing in L_λ the notions of equality and substitution of the simply typed λ-calculus: the rest of the unification process can be viewed as simply an interpreter of L_λ searching for proofs using those axioms.

Chet Murthy

Extracting Proofs from Classical Logic

Abstract

Recently there has been much interest in the problem of finding the computational content of classical reasoning. Work in this area has for the most part focused on re-interpreting various classical results. These different initial starting points result in (ostensibly) completely different notions of computation, spanning the range from concurrent, to sequential algorithms, and yielding semantics of valuations, finite domains, and coherence spaces. In this paper, we attempt to provide a unifying framework for viewing some of these diverse attempts, showing that work of Murthy, and Barbanera and Berardi can be smoothly integrated into the same framework, and seen as different facets of the same phenomenon. We make the first steps toward integrating Girard's classical interpretation into this framework, in the process isolating a new classical reduction, computing under λ-abstraction, which must be added in order to represent his system. We argue that the computational content of a classical proof is most simply understood as a program in an almost-functional language (enriched with Felleisen's "control" operator, $\mathcal{C}$), and that this interpretation yields, under various restrictions, the different methods of extracting computations from classical proofs we examine.

Tobias Nipkow

Order-Sorted Polymorphism

Abstract

ML-style polymorphism can be generalized from a single-sorted algebra of types to an order-sorted one by adding a partially ordered layer of "kinds", "sorts", or "classes" on top of the types. The resulting type system has been implemented in Isabelle to permit type variables in polymorphic constants to range over certain user-definable subsets of all types. It is shown how order-sorted polymorphism allows a simple specification of type restrictions in many logical systems. It can also be used to explain Standard ML's equality types and Haskell's type classes and provides a simple extension of the Hindley/Milner type inference algorithm based on order-sorted unification.

Christine Paulin-Mohring

Properties of Programs extracted from proofs in the Calculus of Inductive Definitions

Abstract

The Calculus of Constructions extended with Inductive Definitions is a suitable framework for the development of certified programs as the computational meaning of constructive proofs. We previously defined for the pure Calculus of Constructions a notion of realizability which interprets a proposition as a set of programs typed in the non-dependent part of the Calculus. In this talk, we shall study the extension of such interpretation to the Inductive Definitions.

Michel Parigot

Free Deduction: an analysis of computations in classical logic

Abstract

Free Deduction (FD) is a deduction system for classical logic designed to understand computations (i.e. cut-elimination) in classical logic - a task which is not achieved by sequent calculus. From a logical viewpoint FD behaves like sequent calculus. But from a computational point of view it allows both a local cut-elimination procedure (like sequent calculus) and a global one (like intuitionistic natural deduction). The main advantage provided by the global elimination procedure is the possibility to state and prove Church-Rosser and Strong Normalisation results for this calculus.

Colin Phillips

Well-founded Induction and Program Synthesis Using Proof Plans

Abstract

The Mathematical Reasoning Group in the Department of Artificial Intelligence at Edinburgh University has developed an approach to the synthesis of programs using proof plans - high level descriptions of proofs which control the automatic search for a proof from which a program can be derived using the proofs as programs technique. The proofs in question employ various forms of induction and techniques have been developed for automating the search for a form of induction appropriate to a given proof.

In this paper I will discuss a modification of this approach which involves using transfinite induction and the automated search for the appropriate well-founded relation. This is being implemented in a version of Martin-Lof's extended by Nordstrom's Acc Type. I shall be comparing this approach with that previously employed in the Mathematical Reasoning Group.

David Pym

Towards an Algebraic Framework for Defining Sequential Logics

Abstract

Intuitionistic type-theoretic logical frameworks are inadequate in several respects:

- They are intuitionistic — weak logics are difficult;
- The notion of logic as proposition-object together with proof-object is present, but too restrictive and not analysed;
- There is no analysis of sequent calculus;
- There is no analysis of proof reduction;
- There is no *clean*, *general* access to a model theory;
- There is no analysis of the *notion* of a framework — characterization of encoding *etc.*.

We propose a framework based on early ideas of Kelly, some recent developments of Hagino, and the hierarchical sensibilities of (Generalized) Type Systems. To this end we propose notions of algebra and dialgebra for indexed functors. Then fibred adjunctions permit the definition of generalizations of Hagino's datatypes and (syntactic) specification language. Starting from an axiomatization of some primitive monoidal connectives, one can obtain, uniformly, Gentzen-type axiomatizations of all such inductively defined logics.

Such a framework provides a quite general analysis of (dependent) type theories, including logically weak theories (*cf.* intuitionistic linear logic-as-linear dependent type theory, considered as sequent calculus together with a notion of proof-object, from the point of view of finitary algebraic presentation. These proposals provide an analysis of the notion of a framework and access to a general notion of Model Theory. One of our (long term) aims is to provide theorems such as: general conditions for the presence of cut-elimination (*cf.* early work of Kelly), natural deduction systems, connection (matrix) calculi, *etc.*. Related work: P. Schroeder–Heister; P. Aczel; E. Moggi; J. Power; C. Wells; E. Ritter; D. Rydeheard.

Jan Smith

An interpretation of Kleene's slash in type theory

Abstract

Kleene introduced the notion of slash to investigate the disjunction and existence properties under implication for intuitionistic arithmetic. In this talk an interpretation of Kleene's slash to type theory is presented. Besides translations of Kleene's results, the main application of the slash in type theory is that conditions are given for a typeable term, containing free variables, to have a normal form beginning with a constructor.

Nora Szasz

A Machine checked proof that Ackermann's function is not primitive recursive

Abstract

A completely machine checked proof of Ackermann's result on the growth of primitive recursive functions is presented. The proof is written in an implementation of Martin-Löf's set theory in ALF. The ALF system was used to carry out the editing and checking of the proof.

List of Participants

Gothenburg & Stockholm

Lennart Augustsson	augustss@cs.chalmers.se
Thierry Coquand	coquand@cs.chalmers.se
Veronica Gaspes	vero@cs.chalmers.se
Clas Lofwall	clas@sics.se
Lena Magnusson	lena@cs.chalmers.se
Per Martin-Löf	
Bengt Nordstrom	bengt@cs.chalmers.se
Kent Petersson	kentp@cs.chalmers.se
Herbert Sander	herbert@cs.chalmers.se
Jan Smith	smith@cs.chalmers.se
Nora Szasz	nora@cs.chalmers.se
Alvaro Tasistro	tato@cs.chalmers.se
Karlis Tcherans	karlis@cs.chalmers.se

Paris 7

Gilles Amiot	amiot@frmap711.bitnet
Vincent Danos	danos@frmap711.bitnet
Jean-Louis Krivine	krivine@frmap711.bitnet
Thierry Lacoste	
Pascale Manoury	eleph@frmap711.bitnet
Patrick Meche	
Michel Parigot	pelz@lri.lri.fr
Christophe Raffalli	raffalli@frmap711.bitnet
Paul Roziere	

INRIA Roquencourt

Gilles Dowek	dowek@margaux.inria.fr
Hugo Herbelin	herbelin@margaux.inria.fr
Gerard Huet	huet@margaux.inria.fr
Yves Lafont	lafont@dmi.ens.fr
Christine Mohring	cpaulin@lip.ens-lyon.fr
Amy Felty	felty@margaux.inria.fr
Benjamin Werner	werner@margaux.inria.fr

INRIA Sophia-Antipolis

Yves Bertot	bertot@mirsa.inria.fr
Thierry Despeyroux	td@mirsa.inria.fr
Gilles Kahn	kahn@mirsa.inria.fr
Laurent Thery	thery@mirsa.inria.fr

Edinburgh

Thorsten Altenkirch	alti@dcs.ed.ac.uk
Paul Anderson	paul@dcs.ed.ac.uk
David Basin	basin@aipna.ed.ac.uk
Gianluigi Bellin	glb@dcs.ed.ac.uk
Alan Bundy	bundy@aipna.ed.ac.uk
Rod Burstall	rb@dcs.ed.ac.uk
Pietro Cenciarelli	pic@dcs.ed.ac.uk
George Cleland	glc@dcs.ed.ac.uk
Philippa Gardner	pag@dcs.ed.ac.uk
Hilf Goguen	hhg@dcs.ed.ac.uk
Claudio Hermida	ch@dcs.ed.ac.uk
Claire Jones	ccmj@dcs.ed.ac.uk
Edmund Kazmierczak	eka@dcs.ed.ac.uk
Monika Lekuse	mkl@dcs.ed.ac.uk
Zhaohui Luo	zl@dcs.ed.ac.uk
James McKinna	jhm@dcs.ed.ac.uk
Savi Maharaj	svm@dcs.ed.ac.uk
Sean Matthews	sean@aipna.ed.ac.uk
Robin Milner	rm@dcs.ed.ac.uk
Eugenio Moggi	em@dcs.ed.ac.uk
Colin Phillips	colinp@aipna.ed.ac.uk
Gordon Plotkin	gdp@dcs.ed.ac.uk
Randy Pollack	rap@dcs.ed.ac.uk
John Power	ajp@dcs.ed.ac.uk
David Pym	dpym@dcs.ed.ac.uk
Don Sannella	dts@dcs.ed.ac.uk
Alex Simpson	als@dcs.ed.ac.uk
Alan Smaill	smaill@dcs.ed.ac.uk

Torino

Franco Barbanera	barba@di.unito.it
Stefano Berardi	stephano@di.unito.it
Mario Coppo	coppo@di.unito.it

Manchester

Peter Aczel	petera@cs.man.ac.uk
Gilles Barthe	
Christian Jelstrup	jelstrup@cs.man.ac.uk
Nax Mendler	mendler@cs.man.ac.uk
David Rydeheard	david@cs.man.ac.uk
Harold Simmons	hsimmons@cs.man.ac.uk

Cambridge

Martin Coen	Martin.Coen@cl.cam.ac.uk
Tom Melham	Tom.Melham@cl.cam.ac.uk
Toby Nipkow	tnn@cl.cam.ac.uk
Larry Paulson	Larry.Paulson@cl.cam.ac.uk

Oxford

Razvan Diaconescu	Razvan.Diaconescu@prg.ox.ac.uk
Joseph Goguen	goguen@prg.ox.ac.uk
Lincoln Wallen	Lincoln.Wallen@prg.ox.ac.uk
David Wolfram	David.Wolfram@prg.ox.ac.uk

Cornell

Robert Constable	rc@gvax.cs.cornell.edu
Chet Murthy	murthy@cs.cornell.edu
Anil Nerode	nerode@msiadmin.cit.cornell.edu

Tel Aviv University

Arnon Avron aa@math.tau.ac.il

Einhoven

De Bruijn wsdwnb@win.tue.nl

St. Andrews

Roy Dyckhoff rd@cs.st-and.ac.uk

CMU

Bob Harper rwh@cs.cmu.edu

Philips

Leen Helmink helmink@prl.philips.nl

Udine

Furio Honsell HONSELL@uduniv.cineca.it

EC

Hussla

Nijmegen

Hermann Geuvers herman@cs.kun.nl

Eindhoven/Nijmegen

Bentham Jutting

RHBNC/LFCS

Ursula Martin uhm@cs.rhbnc.ac.uk

U. Penn/LFCS

Dale Miller dale@linc.cis.upenn.edu

Harvard/INRIA

Gabriel Stolzenberg gabe@zariski.harvard.edu

Liege University

Pierre Wolper pw@montefiore.ulg.ac.be

Witness Extraction in Classical Logic through Normalization

Franco Barbanera Stefano Berardi*
Dipartimento di Informatica
Corso Svizzera, 185
10149 Torino (Italy)

June 1991

Abstract

We show how the strong normalization property for reductions on proofs in classical higher order systems can be used to extract witnesses from proofs of existentially quantified formulas.

1 Introduction

It is an old and well known result that the set of $\sum_1^0$ sentences (and hence, by leaving a free variable, even the one of $\prod_2^0$ sentences) provable in classical logic coincide with that of the sentences provable in intuitionistic logic. The proof of this fact was given by Kreisel in [6]. Later Friedman in [3] showed that Kreisel's difficult proof, which involves functional interpretation, could be condensed down to a simple syntactic argument, his A- translation. By means of this translation one manage to obtain an intuitionistic proof of a $\sum_1^0$ (and then even of a $\prod_2^0$) formula out of a classical proof of the same formula. It is then possible to extract constructive content from classical proofs (if the formulas proved are of the kind above specified) by using the usual extraction algorithms for intuitionistic logic after having performed Friedman's translation.

Recently efforts have been made in order to develop methods to extract *directly* the constructive content from classical proofs. Murthy [7], by building on the work of Griffin [5] and Plotkin [8], has managed to give a consistent algorithmic interpretation to the rule of double negation, the classical axiom, in much the same manner as the other rules of logic are given algorithmic interpretations by constructive realizability. This algorithmic interpretation turns out to be Fellaisen's non-local operator $\mathcal{C}$ [2], a relative of call/cc (call with current continuation) from Scheme [1]. Then by Murthy's result one can employ classical logic as a formal system to deal with non-functional programs using $\mathcal{C}$.

The present paper aims at being a contribution in the direction of developing methods to extract *directly*, without passing trough translations like Friedman's one, the constructive content from classical proofs of $\sum_1^0$ (and $\prod_2^0$) formulas. Our method build's on results of Prawitz which we reformulate and slightly improve. In [9] Prawitz defined reduction rules for natural deduction systems of first order classical and intuitionistic logic and for such reductions gave normalization results. Similar reductions, together with a strong normalization result, were

*barba,stefano@di.unito.it

given by him in [10] even for second order classical and intuitionistic logic. He used the strong normalizazion result in order to infer consistency properties for various logics.

What we show in the present paper is that Prawitz reductions (and the strong normalization for them) can be used for an unespected purpose, i.e. to extract witnesses from classical proofs of existential formulas. The logical system we consider are stronger than the ones considered by Prawitz : we consider all the possible extensions of higher order classical logic by means of *atomic* rules, i.e. the extensions of higher order classical logic by means of *Post Systems*. A particular attention will be given to that particular extension corresponding to higher order Peano Arithmetic (PA^ω).

Our strong normalization result can be viewed as strengthening Prawitz's one because one of the reduction rules we consider is more general that Prawitz's corresponding one, and its application is not postponed until all the other rules are no more applicable as Prawitz did.

Our method of witness extraction works in the following way. If we have a system obtained by adding a Post system to higher order classical logic and in it have a proof of an existentially quantified atomic proposition (recall that for the case of PA^ω each (unary) predicate can be reduced to an atomic proposition of the form $f(x) = 0$), what we do is adding a new rule to the Post system corresponding to the negation of the existentially quantified atomic proposition. It is then possible to obtain a proof of $\perp$ which we put in normal form using the strong normalization result. From a lemma we give about the shape of normal forms we get that the normal form of the so obtained proof of $\perp$ has necessarily to be formed only by rules of the Post system, and then it is easily to get from it the term witnessing the existential formula.

It is worth to point out that this method of witness extraction would work even if we used as reduction rules the ones defined by Statman. Statman's rules would improve the method of extraction from an efficiency point of view and then could be considered in case of an implementation of it.

It is worth to point out that the method of extracting a witness we choosed has a nice and simple interpretation in terms of Kripke semantics and partial valuations. This interpretation is the subject of a forthcoming paper.

Section 2 will consist in a presentation of Post Systems. In Section 3 it will be given a presentation of higher order classical logic extended with Post systems. We shall define, in Section 4, a set of reduction rules on proofs in such higher order systems, and present a strong normalization result, whose proof will be given in the appendix. Section 5 will be devoted to the use of the strong normalization result for extracting witnesses from proofs of existential formulas.

2 Post Systems

In this section we introduce a notion of formal system without logical symbols, but dealing only with atomic propositions and entailment relations among them.
Such systems are known as *Post Systems*. A particular Post system will be the core of the complex systems for classical logic we shall introduce later.

Let us assume to have a countable set of *term variables* $\{ x_0, x_1, x_2, \ldots \}$ and let $\mathcal{F}$ be a set of function symbols, each with its own arity $n \geq 0$.
The set of *terms* over $\mathcal{F}$ is the free algebra built out of the term variables, using $\mathcal{F}$ as set of constructors. Then a term is either a variable or an expression of the form $ft_1 \ldots t_n$ where $f \in \mathcal{F}$, f has arity n and $t_1, \ldots, t_n$ are terms.

In the following terms will be denoted by $t, u, v, \ldots$

Let now $\mathcal{P}$ be a set of predicate symbols, each with its own arity $n \geq 0$.
The *Propositions* in the language of $\mathcal{F}$ and $\mathcal{P}$ are expressions of the form $p(t_1 \ldots t_n)$, with $p \in \mathcal{P}$, p of arity n and $t_1, \ldots, t_n$ terms over $\mathcal{F}$.

Given a set $\mathcal{F}$ of function symbols and a set $\mathcal{P}$ of predicate symbols, an *Atomic Rule* in the language of $\mathcal{F}$ and $\mathcal{P}$ is a pair

$$< \mathbf{x}, \mathbf{A} >$$

where $\mathbf{A} = A_1, \ldots, A_{n+1}$ is a list of $n+1$ propositions of $\mathcal{F}$ and $\mathcal{P}$ and $\mathbf{x} = x_1, \ldots, x_m$ is the list of all the variables in $A_1, \ldots, A_n$.

Definition 2.1 *A* Post System *is a triple*

$$< \mathcal{F}, \mathcal{P}, \Sigma >$$

where $\mathcal{F}$ is a set of function symbols, $\mathcal{P}$ is a set of predicate symbols and Σ is a set of atomic rules in the language of $\mathcal{F}$ and $\mathcal{P}$.

In the following we shall denote a Post system $< \mathcal{F}, \mathcal{P}, \Sigma >$ simply by Σ.
Atomic rules will be denoted by $r, s, \ldots$
In literature atomic rules are sometimes referred to as *Post Productions* or *Horn Clauses*.

Let $r =< \mathbf{x}, \mathbf{A} >$ with $\mathbf{A} = A_1[\mathbf{x}], \ldots, A_n[\mathbf{x}], A_{n+1}[\mathbf{x}]$ and $\mathbf{x} = x_1, \ldots, x_m$.
The intuitive meaning of r is the following : for every list $\mathbf{t} = t_1, \ldots, t_m$,

$$\text{IF} \quad A_1[\mathbf{t}], \ldots, A_n[\mathbf{t}] \quad \text{THEN} \quad A_{n+1}[\mathbf{t}]$$

We introduce now the notion of Atomic Proof. For each atomic proposition A we have a countable set of *proof variables of type* A : $\{x_0^A, x_1^A, x_2^A, \ldots\}$.

Definition 2.2 *Let Σ be a Post system. We say that π is an* atomic proof *for the atomic proposition A, denoted by $\pi : A$, if we can infer $\pi : A$ using the following rules :*

$$x^A : A \qquad\qquad \frac{\pi_1 : A_1[\mathbf{t}] \ldots \pi_n : A_n[\mathbf{t}]}{r(\mathbf{t}, \pi_1, \ldots, \pi_n) : A_{n+1}[\mathbf{t}]}$$

where x^A is any proof variable of type A and $r =< \mathbf{x}, \mathbf{A} >$, with $\mathbf{A} = A_1, \ldots, A_{n+1}$, is any element of Σ.

We shall denote by

$$x_1^{A_1} : A_1, \ldots, x_n^{A_n} : A_n \vdash_\Sigma \pi : A$$

the fact that π is a proof of A in Σ with free variables all in the set $\{ x_1^{A_1}, \ldots, x_n^{A_n} \}$.

We sometimes shall use the notation π^A instead of $\pi : A$.
We shall write

$$A_1, \ldots, A_n \vdash_\Sigma A$$

(to be read $A_1, \ldots, A_n$ entail A in Σ)
if there exists a π such that

$$x_1^{A_1} : A_1, \ldots, x_n^{A_n} : A_n \vdash_\Sigma \pi : A$$

When $n = 0$ we shall also write

$$\Sigma \vdash A$$

(to be read "A is a theorem in Σ").

2.1 An Example of Post System

We introduce now a particular Post system which will play a central role in the following.

Definition 2.3 *We define the Post system* $< \mathcal{F}_0, \mathcal{P}_0, \Sigma_0 >$ *in the following way :*

$\mathcal{F}_0$ = *a set of function symbols, one for each primitive recursive function.*

$\mathcal{P}_0$ = $\{\perp, =\}$ *with* $\perp$ *of arity* 0 *and* $=$ *of arity* 2.

Σ_0 = *one rule for each equation defining a primitive recursive function together with the following rules:*

$<< x >, < 0 = s(x), \perp >>$

$<< x, y >, < s(x) = s(y), x = y >>$

$<< x >, < x = x >>$

$<< \mathbf{x}, x, y >, < x = y, A[x], A[y] >>$ *for each proposition* A *of* $\mathcal{F}_0$ *and* $\mathcal{P}_0$ *.*

Then the Post system $< \mathcal{F}_0, \mathcal{P}_0, \Sigma_0 >$ is a Post system concerning arithmetic of primitive recursive functions. The rules of Σ_0 characterize the primitive recursive functions and state that 0 is not a successor and that the successor function is injective. The last two rules characterize the equality between integers.

3 A Notion of Higher Order Classical System

As already mentioned, we wish to deal with the problem of "witness extraction" not only for higher order arithmetic, but for all the systems extending classical higher order logic with a set of rules which can be described by means of a Post system. Given a Post system Σ containing at least a 0-ary predicate $\perp$(like Σ_0 of the previous section) we extend it to an higher order classical system. Such system is no more than a particular way of presenting the system of higher order classical logic enriched with the rules represented in Σ . We represent proofs as λ-terms (thus in linear form) because in such a way they are easier to "handle".

Let then Σ be any Post system containing a 0-ary predicate symbol $\perp$. We define in what follows a notion of *Higher Order Classical System* C_Σ which extends Σ.

The expressions of the system C_Σ are of four kinds: *types, predicates, terms and proofs.* In the system we have two constants : ι (the type of terms of Σ) and *Prop* (the type of propositions of C_Σ).

The *types* T of C_Σ and a subset of them : the *types of predicates* of C_Σ (T_P) are defined by the following productions:

$$\mathsf{T} ::= \iota | \mathsf{T}_P$$
$$\mathsf{T}_P ::= \mathit{Prop} | \mathsf{T} \to \mathsf{T}_P$$

We shall use T,U,... to denote types.
For each type T of C_Σ We assume to have an infinite number of variables $x_0^T, x_1^T, \ldots$ of type T for each type T of C_Σ .

The *terms* of C_Σ are the terms of Σ.

Let *At* be any proposition of Σ, t a term and T, U types.

The predicates P^{T} of type T of C_Σ is inductively defined in the following way :

$$At^{Prop} \qquad x^T$$

$$(\neg P^{Prop})^{Prop} \qquad (P^{Prop} \wedge P^{Prop})^{Prop}$$

$$(\forall x^T.P^{Prop})^{Prop} \qquad (\lambda x^T.P^U)^{T\to U}$$

$$(P^{T\to U}P^T)^U \qquad (P^{\iota\to T}t)^T$$

The propositions of C_Σ are the predicates of type *Prop*.

From now on we shall consider only predicates that are in $\beta\eta$-normal form, i.e. predicates that contain no subterm of the shape $(\lambda x^T.P)E^T$ or $\lambda x^T.Px^T$ with x^T not free in P. Normal forms may be obtained through a normalization process, whose convergence to a unique normal form may be proved as in simply typed lambda-calculus.
To restrict ourselves to normal predicates, if P is a predicate and E^T an expression (predicate or term) of type T, then $P[x^T := E^T]$ will denote the *normal form* of the term obtained out of P by replacing x by E .

Let us define now the notion of *proof* of a proposition in C_Σ . As stated before, we shall consider only $\beta\eta$-normal propositions.

For each proposition A of C_Σ we have an infinite number of *proof variables* of type A. Let $r =< \mathbf{x}, \mathbf{A} > \in \Sigma$ and E^T be any expression of type T. We say that p is a *proof* for the proposition A, denoted by $p : A$, if we can infer $p : A$ using the following rules :

$$x^A : A \qquad \frac{p_1 : A_1[\mathbf{t}] \ldots p_n : A_n[\mathbf{t}]}{r(\mathbf{t}, p_1, \ldots, p_n) : A_{n+1}[\mathbf{t}]}$$

$$\frac{\begin{array}{c}[x^A : A]\\ \vdots \\ p : \bot\end{array}}{\lambda x : A.p : \neg A} \qquad \frac{\begin{array}{c}[x^{\neg A} : \neg A]\\ \vdots \\ p : \bot\end{array}}{\mathcal{N} x : \neg A.p : A}$$

$$\frac{p_1 : \neg A \quad p_2 : A}{(p_1 p_2) : \bot} \qquad \frac{p : A_1 \wedge A_2}{\pi_i(p) : A_i}$$

$$\frac{p : A}{\lambda x : T.p : \forall x^T.A}(*) \qquad \frac{p : \forall x^T.\mathrm{A}}{(pE^T) : A[x := E]}$$

(*) x not free in B for any proof variable x^B free in p.

Let us informally summarize the main features of C_Σ. The types of C_Σ are built out of the constants ι and *Prop* using the arrow ($\to$). We have forbidden, for sake of simplicity, types where ι occurs on the righthand side of $\to$, like $\iota \to \iota$, $Prop \to \iota, \ldots$ Type of predicates are those of the form $T_1 \to \ldots \to T_n \to Prop$, i.e. they are classes of predicates with arguments in $T_1 \ldots T_n$. The only other type is the type ι of terms of Σ.

The propositions of C_Σ are either atomic propositions of Σ, or negation or conjunction of propositions, or universal quantifications over objects, or expressions of the shape $P(E_1, \ldots, E_n)$ where P is a predicate. Predicates are either propositions or variables or λ-abstractions or applications. Predicates should be identified up to $\beta\eta$-conversion. We did it by considering only $\beta\eta$-normal predicates.

The rules of the system are the rules of Σ, introduction and elimination rules for $\neg$, $\wedge$, $\forall$ and double negation.

We shall denote by

$$x_1^{A_1} : A_1, \ldots, x_n^{A_n} : A_n \vdash_{\mathsf{C}_\Sigma} p : A$$

the fact that p is a proof of A in C_Σ with free variables all in the list $x_1^{A_1}, \ldots, x_n^{A_n}$.

We sometimes shall use the notation p^A instead of $p : A$.

We shall write

$$A_1, \ldots, A_n \vdash_{\mathsf{C}_\Sigma} A$$

(to be read $A_1, \ldots, A_n$ entail A in C_Σ), if there exists a p for which

$$x_1^{A_1} : A_1, \ldots, x_n^{A_n} : A_n \vdash_{\mathsf{C}_\Sigma} p : A$$

When $n = 0$ we shall also write

$$\mathsf{C}_\Sigma \vdash A$$

(to be read A is a theorem in C_Σ).

3.1 An Example of Classical System : Higher Order Peano Arithmetic

In this subsection we show that Higher Order Peano Arithmetic (PA^ω) can be presented as a classical higher order system.

Let Σ_0 be the Post system defined in the previous section. If we consider the classical system C_{Σ_0}, in it we have the language of arithmetic, atomic axioms expressing basic facts about integers and higher order classical logic. To show that C_{Σ_0} is exactly PA^ω it is enough to explain how to express the induction rule in the former.

Before doing it a few definitions are necessary, i.e. how to encode natural numbers and the logical connectives $\to$ and $\exists$.

$$\begin{array}{lll}
A \to B & =_{def} & \neg(A \wedge \neg B) \\
\mathsf{N} & =_{def} & \lambda n : \iota.\forall X : \iota \to Prop.X(0) \to (\forall x : \iota.X(x) \to X(s(x))) \to X(n) \\
\forall x \in \mathsf{N}.A & =_{def} & \forall x : \iota.\mathsf{N}(x) \to A \\
\exists x \in \mathsf{N}.A & =_{def} & \exists x : \iota.\mathsf{N}(x) \wedge A
\end{array}$$

Let P be a predicate of type $\iota \to Prop$ in PA^ω. We can express the induction rule on P in the following way:

$$Ind[P] =_{def} \forall x \in \mathsf{N}.P[0] \to (\forall y \in \mathsf{N}.P[y] \to P[s(0)]) \to P[x]$$

Lemma 3.1 $\mathsf{C}_{\Sigma_0} \vdash \forall X : \iota \to$ *Prop.Ind[X].*

Proof. Left to the reader. It is useful to check first that $A, A \to B \vdash_{\mathsf{C}_{\Sigma_0}} B, \mathsf{C}_{\Sigma_0} \vdash \mathsf{N}(0)$ and $\mathsf{C}_{\Sigma_0} \vdash \forall x : \iota.\mathsf{N}(x) \to \mathsf{N}(s(x))$.$\Box$

We have thus shown that C_{Σ_0} is indeed PA^ω.

4 Strong Normalization for Classical Proofs

In this section we introduce a set of reduction rules on proofs in classical systems. These rules are the usual Prawitz's reduction rules for intuitionistic logic and four reduction rules dealing with double negation [9,10]. For these rules we shall give a result of strong normalization. In the next section we shall show that such a result, besides implying consistency result, provides also a mean to extract witnesses from existential proofs.

The reduction rules for intuitionistic logic transform a proof by eliminating consecutive applications of introduction and elimination rules for the same connective.

Definition 4.1 *The reduction rules for intuitionistic logic are the following :*

¬-int) $(\lambda x^A.p^\perp)^{\neg A}(q^A) \quad \to \quad p^\perp[x := q^A]$

∧-int) $\pi_i < p_1^{A_1}, p_2^{A_2} > \quad \to \quad p_i^{A_i}$

∀-int) $(\lambda x^T.p^A)^{\forall x^T.A}(E^T) \quad \to \quad p^A[x^T := E^T]$

In order to define the reduction rules dealing with double negation, we have first to introduce standard proofs of

$$A \vdash_{\mathsf{C}_\Sigma} \neg\neg A \qquad \neg A_i \vdash_{\mathsf{C}_\Sigma} \neg(A_1 \wedge A_2) \qquad \neg A[E] \vdash_{\mathsf{C}_\Sigma} \neg\forall x^T.A$$

Let $p^{\neg A}$, $p^{\neg A_i}$, $p^{\neg A[E]}$ be proofs, and x, y, z, c fresh variables.

$$\sigma[p^A] =_{def} (\lambda y^{\neg A}.(y^{\neg A}p^A)^\perp)^{\neg\neg A}$$

$$\sigma_i[p^{\neg A_i}] =_{def} (\lambda z^{A_1\wedge A_2}.(p^{\neg A_i}\pi_i(z)^{A_i})^\perp)^{\neg(A_1\wedge A_2)}$$

$$\sigma_E[p^{\neg A_E}] =_{def} (\lambda c^{\forall x.A}.(p^{\neg A_E}c^{\forall x.A}(E)^{A_E})^\perp)^{\neg\forall x.A}$$

We may now define the reduction rules dealing with double negation. These rules transform a proof in order to have applications of the rule of double negation performed on simpler and simpler formulas. If a proof is normal with respect these reductions then double negation is applied only on atomic formulas.

Definition 4.2 *The reduction rules for application of double reduction rules are the following:*

¬-class $(\mathcal{N}x^{\neg\neg A}.p[x]^\perp)^{\neg A} \quad \to \quad (\lambda x^A.p[\sigma[x]]^\perp)^{\neg A}$

∧-class $(\mathcal{N}x^{\neg(A_1\wedge A_2}.p[x]^\perp)^{A_1\wedge A2} \quad \to \quad < q_1^{A_1}, q_2^{A_2} >^{A_1\wedge A_2}$ *with* $q_i^{A_i} = (\mathcal{N}x^{\neg A_i}.p[\sigma_i[x]]^\perp)^{A_i}$

∀-class $(\mathcal{N}x^{\neg\forall y^T.A}.p[x]^\perp)^{\forall y^T.A} \quad \to \quad (\lambda y^T.q^A)^{\forall y^T.A}$ *with* $q^A = (\mathcal{N}x^{\neg A}.p[\sigma_y[x]]^\perp)^A$

At-**class** $p^{At}[q^{At}] \quad \to \quad q^{At}$ *where At is any atomic proposition of* Σ *and* $p^{At}[q^{At}]$ *is a proof of At containing a proper subproof q of At such that FV(q)* $\subseteq$ *FV(p).*

In order to clarify the sort of transformation performed by the rules for double negation given above we give also their version in natural deduction.

$$\begin{array}{c} [\neg\neg A] \\ \vdots \\ \perp \\ \hline \neg A \end{array} \quad \rightsquigarrow \quad \begin{array}{c} \dfrac{\dfrac{[\neg A] \qquad [A]}{\perp}}{\neg\neg A} \\ \vdots \\ \perp \\ \hline \neg A \end{array}$$

$$\dfrac{\begin{array}{c}[\,\neg A_1 \wedge A_2\,]\\ \vdots\\ \bot\end{array}}{A_1 \wedge A_2} \quad \leadsto \quad \dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{[\neg A_1] \qquad \dfrac{[A_1 \wedge A_2]}{A_1}}{\bot}}{\neg A_1 \wedge A_2}\\ \vdots\\ \bot\end{array}}{A_1} \qquad \dfrac{\begin{array}{c}\dfrac{\dfrac{[\neg A_2] \qquad \dfrac{[A_1 \wedge A_2]}{A_2}}{\bot}}{\neg A_1 \wedge A_2}\\ \vdots\\ \bot\end{array}}{A_2}}{A_1 \wedge A_2}$$

$$\dfrac{\begin{array}{c}[\neg \forall x.A[x]]\\ \vdots\\ \bot\end{array}}{\forall x.A[x]} \quad \leadsto \quad \dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{[\neg A[y]] \qquad \dfrac{[\forall x.A[x]]}{A[x]}}{\bot}}{\neg \forall x.A[x]}\\ \vdots\\ \bot\end{array}}{A[y]}}{\forall y.A[y]}$$

The reduction (At-class) is more general of the similar one considered by Prawitz [10]. Given the reduction rules we can now define the reduction relations among proofs induced by them.

Definition 4.3 *Given two proofs p and q we have that $p \longrightarrow q$ if q is obtained by replacing one redex in p by its contractum.*
$\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$

The reduction relation $\longrightarrow^*$ has several familiar properties. For instance :

Lemma 4.1 *(i) If $\mathbf{x} : \mathbf{A} \vdash p : A$ and $p \longrightarrow^* p'$, then $\mathbf{x} : \mathbf{A} \vdash p' : A$ (Subject Reduction property). (ii) If $p[\mathbf{x}] \longrightarrow^* q[\mathbf{x}]$ and $\mathbf{E}$ is a list of expressions having the same types as $\mathbf{x}$, then $p[\mathbf{E}] \longrightarrow^* q[\mathbf{E}]$.*

Proof Left to the reader. Essentially the lemma holds because the classes of the propositions starting with $\neg, \wedge, \forall$ and of the propositions of Σ are all closed by substitution. □

The reduction relation $\longrightarrow^*$ defined above is clearly not confluent because of the presence of the rule At-class). Therefore a proof can have several different normal forms. What we shall prove however is that $\longrightarrow^*$ is strongly normalizing.

Theorem 4.1 *The reduction relation $\longrightarrow^*$ is strongly normalizing, i.e. for any proof p there always exists a finite bound n to the length of any reduction path $p \longrightarrow^* p_1 \longrightarrow^* p_2 \longrightarrow^* \ldots$*

The proof of this theorem will be given in the Appendix.

It is worth noting that this result of strong normalization is stronger than the one proved by Prawitz [9,10], since he considered only a particular case of our rule At-class) and since in [10] the reductions on atomic proofs can be performed only at the very end of the reduction path, when no other reduction is possible. Our strong normalization result then holds even when these restrictions are taken out.

5 Witness Extraction in Classical Logic

In this section we show that the strong normalization for the reductions on proofs has an unexpected use:

> *if $PA^\omega \vdash p : \exists x \in \mathsf{N}.P[x]$ and P is a decidable predicate, then it is possible to extract in an effective way from p an integer n such that $PA^\omega \vdash P[n]$.*

Indeed it is possible to have something stronger, i.e. if we have $\mathsf{C}_\Sigma \vdash \exists x.At[x]$ (with At proposition of Σ) then it is possible to find in an effective way a term t such that $\mathsf{C}_\Sigma \vdash At[t]$. We shall consider for the witness extraction the case of PA^ω for its interest. The generalization for the case of any C_Σ is straightforward.

The restriction about P (i.e. P is a decidable predicate) given for the case PA^ω is useful because, if we consider predicates where the only predicate symbol is "=" (which is decidable), we can always put them in the form $f(x) = 0$.

Let Σ be any Post system containing at least a 0-ary predicate symbol $\perp$. We shall first prove a technical lemma about normal proofs in C_Σ and then we use it to derive Prawitz's equiconsistency result for Σ and C_Σ. We then explain how such result can be used to extract a witness from a proof.

Lemma 5.1 *Let p be a normal proof in C_Σ and $\mathbf{At} = At_1, \ldots, At_n$ be any sequence of propositions of Σ ($\neg\mathbf{At}$ denotes $\neg At_1, \ldots, \neg At_n$). Then*
(i) If A is not atomic and $\mathbf{x} : \mathbf{At} \vdash_{\mathsf{C}_\Sigma} p : A$ then p is $x_i^{\neg At_i}$ for some i or p ends with an introduction.
(ii) If At is any proposition of Σ and $\mathbf{x} : \mathbf{At} \vdash_{\mathsf{C}_\Sigma} p : At$ then p contains only rules of Σ, double negations on propositions of Σ or expressions of the form $x^{\neg B}q^B$ with x free or bound and B atomic.
(iii) If $\mathsf{C}_\Sigma \vdash p : \perp$ then $\Sigma \vdash p : \perp$.

Proof. (i) By induction on the structure of p, distinguishing the various cases according to the outermost symbol of p (the last rule used to form p in Natural Deduction style).

- If p is a variable we have finished.
- p cannot be $r(\mathbf{t}, p_1, \ldots, p_n)$ because A is not atomic.
- If p ends with an introduction we have finished.
- If p ends with an elimination then p is either $(q_1^{\neg A} q_2^A)^\perp$ or $(\pi_i q^{A_1 \wedge A_2})^{A_i}$ or $(q^{\forall x.A} E)^{A[E]}$. The first subcase leads to a contradiction, since A is not atomic and cannot be $\perp$. In the remaining subcases we have to note that $q^{A_1 \wedge A_2}$ and $q^{\forall x.A}$ cannot be variables because in such a case they would be $x^{\neg At_i}$ for some i and $A_1 \wedge A_2$ and $\forall x.A \neq \neg At_i$. Then we can apply the induction hypothesis on $q^{A_1 \wedge A_2}$ and $q^{\forall x.A}$ obtaining that $q^{A_1 \wedge A_2} \equiv < q_1^{A_1}, q_2^{A_2} >$ and $q^{\forall x.A} \equiv \lambda x.q_1^A$. This forces p not to be normal, contradiction.
- If p is $(\mathcal{N} x^{\neg A}.q^\perp)^A$ then, since A is not atomic, p would not be normal.

(ii) By induction on the structure of p.

- p cannot be a variable because it is a proof of At and the variables are $x_1^{\neg A_1}, \ldots, x_n^{\neg A_n}$.

- If p is $r(\mathbf{t}, p_1, \ldots, p_n)$ we have finished by induction hypothesis on $p_1, \ldots, p_n$.
- p cannot end with an introduction since otherwise At would be a composite formula.
- If p ends with an elimination then p is either $(q_1^{\neg A} q_2^A)^\perp$ or $(\pi_i q^{A_1 \wedge A_2})^{A_i}$ or $(q^{\forall x.A} E)^{A[x:=E]}$. By applying (i) to $q_1^{\neg A}, q^{A_1 \wedge A_2}$ and $q^{\forall x.A}$ we have that such proofs are either $x_i^{\neg A_i}$ for some i or they end with an introduction. The first subcase can apply only to $q_1^{\neg A}$. From $q_1^{\neg A} \equiv x_i^{\neg At_i}$ we deduce that $p \equiv (x_i^{\neg At_i} q_2^A)^\perp$ and then the thesis follows by the induction hypothesis on q_2^A. In the second subcase we would have $q^{\neg A} \equiv \lambda x^A.q'^\perp$, $q^{A_1 \wedge A_2} \equiv < q_1^{A_1}, q_2^{A_2}$. and $q^{\forall x.A} \equiv \lambda x.q'^A$ and then p would not be normal, contradiction.
- If p is $(\mathcal{N} x^{\neg At}.q^\perp)^{At}$ we have that $\mathbf{x} : \neg \mathbf{At}, x : \neg At \vdash_{\mathsf{C}_\Sigma} q^\perp : \perp$ and then we have finished by the induction hypothesis on $q^\perp$.

(iii) Let us assume that $\mathsf{C}_\Sigma \vdash p : \perp$. We want to prove $\Sigma \vdash p : \perp$, i.e. that p consists only of rules of Σ. By (ii) p consists only of rules of Σ, double negations on atomic propositions and of expressions of the form $x^{\neg B} q^B$. Let us assume, by contradiction, that there is some subterm of p of the form $(\mathcal{N} x^{\neg B} q^\perp)^B$ or $x^{\neg B} q^B$ for some B of Σ. In the first case we distinguish two subcases:

1. for some $\mathcal{N} x.q$ in p we have $x \in FV(q)$
2. for all $\mathcal{N} x.q$ in p we have $x \notin FV(q)$.

In subcase 1, let $(\mathcal{N} x^{\neg B}.q^\perp)^B$ be a subproof of p minimal among the subproofs $\mathcal{N} x'.q'$ such that $x' \in FV(q')$. Thus for each $\mathcal{N} x'.q'$ inside q we have $x' \notin FV(q')$. By (ii) each occurrence of $x^{\neg B}$ in $q^\perp$ is in some expression of the form $(x^{\neg B} q''^B)^\perp$. Let us choose now a minimal $(x^{\neg B} q''^B)^\perp$ in $q^\perp$. Then $x^{\neg B} \notin FV(q''^B)$. Hence $FV(q''^B) \subseteq FV(\mathcal{N} x.q) \cup \{$ variables bound by some $\mathcal{N}$ in q $\}$. Since for each $\mathcal{N} x'.q'$ occurring in q we have $x' \notin FV(q')$, we deduce that $FV(q''^B) \subseteq FV(\mathcal{N} x.q)$. This contradicts the fact that p is normal, because $(\mathcal{N} x.q)^B \longrightarrow q''^B$.
In subcase 2, for all $(\mathcal{N} x^{\neg B}.q^\perp)^B$ occurring in $p^\perp$ we have that $x^{\neg B} \notin q^\perp$. Thus no variable, free or bound , occurs in p. Therefore $FV(q) = FV(p) = \emptyset$ and then $p^\perp \longrightarrow q^\perp$, contradicting the fact that p is normal.
There is still left to derive a contradiction in the second case, when some $(x^{\neg B} q^B)^\perp$ occurs in $p^\perp$. Since $FV(p^\perp) = \emptyset$, $x^{\neg B}$ must be bound by some $(\mathcal{N} x^{\neg B}.q^\perp)^B$ in p. Thus we are in the first case and have finished. □

From point (iii) of the Lemma above and from the strong normalization Theorem it is easy to derive Prawitz's equiconsistency result.

Theorem 5.1 (Prawitz) *For any Post system containing* $\perp$ *:*

$$\Sigma \vdash \perp \Leftrightarrow \mathsf{C}_\Sigma \vdash \perp.$$

Proof. If $\Sigma \vdash p : \perp$ then $\mathsf{C}_\Sigma \vdash p : \perp$ because Σ is a subsystem of C_Σ. If $\mathsf{C}_\Sigma \vdash p : \perp$ then, by the strong normalization Theorem, there exists a normal form p' of p. Hence $\mathsf{C}_\Sigma \vdash p' : \perp$ and then, by point (iii) of the previous lemma, $\Sigma \vdash p' : \perp$.□

Prawitz used this result to infer the consistency of $PA^\omega (PA^\omega \not\vdash \perp)$ from the consistency of Σ_0, which it is easy to prove.
We show now how Prawitz's result can have a computational use.

Let us assume to have

$$PA^\omega \vdash p : \exists x \in \mathsf{N}.(f(x) = 0).$$

By "expanding" the abbreviations used we have then

$$PA^{\omega} \vdash p : \neg\forall x : \iota.\neg(\mathsf{N}(x) \wedge (f(x) = 0))$$

Let now

$$r = << x >, < (f(x) = 0), \perp >>$$

be a fresh atomic rule. In natural deduction style this rule would have the following shape :

$$r)\,\frac{f(x) = 0}{\perp}.$$

The central remark is now that $PA^{\omega} + r \vdash \perp$. In fact, if we put

$$q =_{def} (\lambda x^{\mathsf{N}(x)\wedge(f(x)=0)}.r(\pi_2(x)^{(f(x)=0)})^{\perp}$$

then

$$PA^{\omega} + r \vdash \lambda x.q : \forall x.\neg(\mathsf{N}(x) \wedge (f(x) = 0))$$

We can then obtain

$$PA^{\omega} + r \vdash p(\lambda x.q) : \perp.$$

Put in natural deduction style the proof $p(\lambda x.q)$ has the following shape:

$$\dfrac{\begin{array}{c}\vdots\\ \neg\forall x.\neg(\mathsf{N}(x) \wedge (f(x) = 0))\end{array} \qquad \dfrac{\dfrac{\dfrac{\dfrac{[\mathsf{N}(x) \wedge (f(x) = 0)]}{f(x) = 0}}{\perp}}{\neg(\mathsf{N}(x) \wedge (f(x) = 0))}}{\forall x.\neg(\mathsf{N}(x) \wedge (f(x) = 0))}}{\perp}$$

By the proof of Prawitz's equiconsistency result, if p' is any normal form of $p(\lambda x.q)$, we have that

$$\Sigma_0 + r \vdash p' : \perp.$$

No *proper* subterm of type $\perp$ can occur in p' because p' is in normal form. On the other hand r may occur in p' only in subproofs of the form $r(t, q')$ having type $\perp$. Moreover r must occur in p' since $\Sigma_0 \not\vdash \perp$.
Thus p' itself is of the form $r(t, q')$ for some term t of Σ_0 and r does not occur in q'. The subproof q' must have the type (f(t)=0). Since r does not occur in q' we have

$$\Sigma_0 \vdash q' : (f(t) = 0)$$

Then it is easy to generate, by means of a normalization process, from a proof in higher order Peano arithmetic of $\exists x \in \mathsf{N}.f(x) = 0$, a term t such that $f(t) = 0$.

It is worth noting that because of the presence of the rule (At-class) (and hence of the non-confluence of the reductions) it is possible to get different witnesses by following different reduction strategies. This is not possible in intuitionistic logic, where the witness that it is possible to "extract" from a proof is unique.

6 Appendix : Strong Normalization for the Systems C_Σ

In this appendix we give the proof of the result claimed in section 4. i.e. that every proof p in a system $\mathcal{C}_\Sigma$ is strongly normalizing under the set of reduction rules defined.

Since the argument is quite complex, we shall split it in four subsections. In 6.1 we shall introduce, using Girard's candidates [4], a Realization Semantic for the intuitionistic propositional part of $\mathcal{C}_\Sigma$; in 6.2 we shall extend such semantic to the classic propositional part of $\mathcal{C}_\Sigma$; this semantic will be further extended to the first and higher order part of $\mathcal{C}_\Sigma$ in 6.3. The strong normalization result will be proved in 6.4.

6.1 Realization Semantic for the Intuitionistic Propositional Fragment of C_Σ

In this subsection we introduce the notion of Girard's candidate and prove that it can generate a Realization semantic for Σ, the connectives $\neg, \wedge$ and the intuitionistic rules for them.

Let us firstly give two abbreviations to be used later. Let A be a proposition and p a proof in $\mathcal{C}_\Sigma$.

$$n \textit{ is a bound for } p \text{ iff the reduction tree of } p \text{ has a finite height} \leq n.$$

$$SN_A =_{def} \{q | q \text{ is a proof of } A \text{ for which there exists a bound } \}$$

We may now define Girard candidates.

Definition 6.1 *Let A be a proposition and C a set of proofs. C is a Girard candidate for A if the following conditions are satisfied :*

cand 1 $C \subseteq SN_A$

cand 2 *If $p \in C$ then for all proofs q,*

$$(p \longrightarrow q) \Rightarrow q \in C$$

cand 3 *If $p \in C$, p does not start with an introduction symbol and for all proofs q of A we have that $(p \longrightarrow q) \Rightarrow q \in C$, then $p \in C$.*

If C is a candidate for A, then $x^A \in C$ by (cand 3), just because $x^A \longrightarrow q$ never holds.

Let us introduce two more abbreviations.

$$\mathbf{GC} =_{def} \{C | C \text{ is a Girard candidate } \}$$

Let C be a candidate for A. Then

$$C^* =_{def} A$$

C^* is unique. If C is a candidate for A and B then by (cand 1) and (cand 3) we have that $x^A \in C \subseteq SN_B$ and thus $A = B$.

We prove now that SN_A is a candidate, and that it is the only candidate for A if A is atomic.

Lemma 6.1 *(i) $SN_A \in \mathbf{GC}$.*
(ii) If $C \in \mathbf{GC}$ and $C^* = A$ with A atomic then $C = SN_A$.

Proof. (i) SN_A satisfies (cand 1) because $SN_A \subseteq SN_A$. It satisfies (cand 2) because if $p \in SN_A$ and $p \longrightarrow q$ then $q \in SN_A$. Suppose now that for all proofs q, $p \longrightarrow q \Rightarrow q \in SN_A$. There are only finitely many q's such that $p \longrightarrow q$, say $q_1, \ldots, q_m$. Let n_i be a bound for $q_i, i = 1, \ldots, m$. Now $\max_i\{n_i + 1\}$ is a bound for p and hence $p \in SN_A$. Thus SN_A satisfies (cand 3).
(ii) Let $C \in \mathbf{GC}$ and $C^* = A$ with A atomic. By (cand 1) we have only to prove that $SN_A \subseteq C$. Let then $p \in SN_A$ and n be a bound for p. We prove $p \in C$ by induction on n. Remark that p cannot begin with an introduction, otherwise A would be composite. Therefore, by (cond 3), it is enough to prove that for all proofs q, $p \longrightarrow q \Rightarrow q \in C$. If $p \longrightarrow q$ then q has a bound $n' < n$ and , by the induction hypothesis on n' we have that $q \in C$. $\Box$

Thus, if A is atomic, we may canonically interpret it by SN_A. We prove now that such "interpretation" is sound with respect to the rules of Σ.

Lemma 6.2 *Let* $r =< \mathbf{x}, < A_1, \ldots, A_{n+1} >> \in \Sigma$ *and* $C_1, \ldots, C_{n+1} \in \mathbf{GC}$ *such that* $C_i^* = A_i, i = 1, \ldots, n+1$. *Then*

$$p_1 \in C_1, \ldots, p_n \in C_n \Rightarrow r(\mathbf{t}, p_1, \ldots, p_n) \in C_{n+1}$$

Proof. Let in this proof p be short for $r(\mathbf{t}, p_1, \ldots, p_n) \in C_{n+1}$. Since $C_i \subseteq SN_{A_i}$, $p_1, \ldots, p_n$ have bounds, say $k_1, \ldots, k_n$. By induction on$\sum_{i=1}^n k_i$ we prove that $p \in C_{n+1}$. Since p begins with r, by (cand 3) it is enough to prove that $p' \in C$ for all p' such that $p \longrightarrow p'$. A reduction on p can be performed either inside some p_i or can be (At-class), with p itself as redex. In the first case we deduce $p' \in C_{n+1}$ by induction on $\sum_{i=1}^n k_i$ and the closure of C_i under $\longrightarrow$. In the second case p' is a subproof of some p_j and therefore $p' \in SN_{A_{n+1}}$. By point (ii) of the previous lemma $C_{n+1} = SN_{A_{n+1}}$, thus $p' \in C_{n+1}$.$\Box$

We interpret now the connectives $\neg$ and $\wedge$ as operations on candidates.
Let us give two more abbreviations.

$$C_1 \underline{\wedge} C_2 =_{def} \{p | p \text{ is a proof and } \pi_i(p) \in C_i, i = 1, 2\}$$

$$\underline{\neg} C =_{def} \{p | p \text{ is a proof and } \forall a \in pa \in SN_\bot\}$$

By the consistency of the typing for $\pi_i(p), pa$, we deduce $(C_1 \underline{\wedge} C_2)^* \equiv C_1^* \wedge C_2^*$ and $(\underline{\neg} C)^* \equiv \neg C^*$.

We check now that $\wedge$ and $\neg$ indeed build new elements of $\mathbf{GC}$.

Lemma 6.3 *(i)* $\underline{\wedge} : \mathbf{GC} \to \mathbf{GC} \to \mathbf{GC}$ *(ii)* $\underline{\neg} : \mathbf{GC} \to \mathbf{GC}$.

Proof. (i) Let $C_1, C_2 \in \mathbf{GC}$ and $C_i^* = A_i, i = 1, 2$. To prove (cand 1), if $p \in C_1 \underline{\wedge} C_2$ we have that $\pi_i(p) \in C_i \subseteq SN_{A_i}$. Thus $p \in SN_{A_1 \wedge A_2}$. To prove (cand 2), let us suppose $p \in C_1 \underline{\wedge} C_2$ and $p \longrightarrow q$. Then $\pi_i(p) \in C_i$ and $\pi_i(p) \longrightarrow \pi_i(q)$. Thus $\pi_i(q) \in C_i$ and therefore $q \in C_1 \underline{\wedge} C_2$. To prove (cand 3), let us assume that p does not begin with an introduction and that for all proofs q, $(p \longrightarrow q) \Rightarrow q \in C_1 \underline{\wedge} C_2$. First of all we remark that $p \in SN_{A_1 \wedge A_2}$. This follows from (cand 3) for $SN_{A_1 \wedge A_2}$ and from the fact that for all proofs q, $(p \longrightarrow q) \Rightarrow q \in C_1 \underline{\wedge} C_2 \subseteq SN_{A_1 \wedge A_2}$. Now we have to prove that $\pi_i(p) \in C_i$. Since $\pi_i(p)$ begins with π_i, by (cand 3) it is enough to prove that, for all proofs q', $(\pi_i(p) \longrightarrow q') \Rightarrow q' \in C_i$. Since p does not begins with a $< -, - >$, $\pi_i(p)$ is not a π-redex. Thus q' is either $\pi_i(q)$ for some q such that $p \longrightarrow q$ or it is a subproof of p and we applied a reduction (At-class) and A_i must be atomic. In the first case, since $q \in C_1 \underline{\wedge} C_2$ by hypothesis, we conclude that $q' \equiv \pi_i(q) \in C_i$. In the second case $q' \in SN_{A_i}$ because $p \in SN_{A_1 \wedge A_2}$ and q' is a subproof of p. Since A_i is atomic, by Lemma 6.1(ii) we have

that $C_i = SN_{A_i}$. We then conclude $q' \in C_i$.
(ii) Let $C \in \mathbf{GC}$ and $C^* = A$. To prove (cand 1) we remark that if $p \in \neg C$ then, since $x^A \in C$, $px^A \in SN_\perp$ and thus $p \in SN_\perp$. To prove (cand 2) we assume $p \in \neg C$ and want to prove that, if $p \longrightarrow p'$, we have $p'a \in SN_\perp$ for all $a \in C$. By definition of $\neg C$, $pa \in SN_\perp$. Since $pa \longrightarrow p'a$ we conclude $p'a \in SN_\perp$. To prove (cand 3) we assume that p does not begin with an introduction and that, for all proofs p', $(p \longrightarrow p') \Rightarrow p' \in \neg C$. Since $a \in C \subseteq SN_\perp$, we can proceed by induction on the bound n of a. We want to prove $pa \in SN_\perp$ for all $a \in C$. First of all we remark that $p \in SN_{\neg A}$. This follows by (cand 3) for $SN_{\neg A}$ and by the fact that, for all proofs q', $(p \longrightarrow p' \Rightarrow p' \in \neg C \subseteq SN_{\neg A}$. Since pa begins with an application, by (cand 3) for $SN_\perp$ it is enough to prove that for all proofs b, $(pa \longrightarrow b) \Rightarrow b \in SN_\perp$. p does not begin with an abstraction, so pa is not a λ-redex. Thus the reduction can take place in p, a or can be (At-class). If $p \longrightarrow p'$, then by the induction hypothesis $p' \in \neg C$ and therefore $p'a \in SN_\perp$. If $a \longrightarrow a'$ then we have finished by the induction hypothesis and by the fact that $a' \in C$ (which follows by (cand 2). If we applied rule (At-class) then b is a subproof of either p or a. Since both p and a strongly normalize, then $b \in SN_\perp$. □

Thus candidates soundly interpret the formation of intuitionistic quantifier-free propositions. We prove now that they soundly interpret the intuitionistic rules for such propositions.

Lemma 6.4 *Let $C_1, C_2, C \in \mathbf{GC}$ and $C_i^* = A_i, C^* = A$.*
(i) $p_1 \in C_1, p_2 \in C_2 \quad \Rightarrow \quad <p_1, p_2> \in C_1 \wedge C_2$
(ii) $p \in C_1 \wedge C_2 \quad \Rightarrow \quad \pi_i(p) \in C_i, i = 1, 2$
(iii) $\forall a \in C\ p[x := a] \quad \Rightarrow \quad \lambda x.p \in \neg C$
(iv) $p \in \neg C, a \in C \quad \Rightarrow \quad pa \in SN_\perp$

Proof. (i) We have to prove $\pi_i <p_1, p_2> \in C_i$. Since $C_i \subseteq SN_{A_i}$, p_1 and p_2 have bounds, say n_1 and n_2. Therefore we may proceed by induction on $n_1 + n_2$. Since $\pi_i <p_1, p_2>$ begins with π_i, by (cand 3) it is enough to prove that, for all proofs q, $(\pi_i <p_1, p_2> \longrightarrow q) \Rightarrow q \in C_i$. If the reduction on $\pi_i <p_1, p_2>$ takes place inside some p_i, then $q \in C_i$ by the induction hypothesis and the closure of C_i under $\rightarrow$. If $\pi_i <p_1, p_2>$ itself is the redex being reduced, then either $\pi_i <p_1, p_2> \longrightarrow p_i$ or we apply a reduction (At-class) and A_i is a proposition of Σ and q is inside some p_i. In the first case $p_i \in C_i$ by the hypothesis and then we have finished. In the second case $q \in SN_{A_i}$ because $p_i \in SN_{A_i}$. By Lemma 6.1(ii), since A_i is atomic, we have $C_i = SN_{A_i}$ and can conclude $q \in C_i$.
(ii) If $p \in C_1 \underline{\wedge} C_2$ then $\pi_i(p) \in C_i$ by definition of $\underline{\wedge}$.
(iii) Let us assume that $\forall a \in C\ p[x := a] \in SN_\perp$. We want to prove $\forall a \in C\ (\lambda x.p)a \in SN_\perp$. Since $x^A \in C$ then $p \equiv p[x := x^A] \in SN_\perp$. Moreover, $a \in C \subseteq SN_A$. Thus p, a have bounds n, m. We prove $(\lambda x.p)a \in SN_\perp$ by induction on $n + m$. Since $(\lambda x.p)a \in SN_\perp$ begins with an elimi-nation of $\neg$, by (cand 3) it is enough to prove that for all proofs q, $((\lambda x.p)a \longrightarrow q) \Rightarrow q \in SN_\perp$. If the reduction takes place inside p or a we can deduce that $q \in SN_\perp$ by the induction hypothesis, by the closure of C under $\rightarrow$ and by the compatibility of reductions with substitution (Lemma 4.1(ii)). If $(\lambda x.p)a$ itself is the redex being reduced, then either $(\lambda x.p)a \longrightarrow p[x := a]$ or we perform a reduction (At-class) and q is inside p or a. In the first case $p[x := a] \in SN_\perp$ by the hypothesis and we have finished. In the second case, since p, a have a bound, $q \in SN_\perp$ again. □

The interpretation we have outlined for the quantifier-free fragment of $\mathcal{C}_\Sigma$ is not necessarily sound with respect to the double negation rule.

In the next section we shall show that, by considering a suitable subset of Girard candidates, we can also interpret the rule of double negation.

6.2 Adding the Double Negation Rule

In this subsection we introduce a subset of Girard candidates (the Prawitz Candidates), which are exactly those which soundly interpret the double negation rule. Then we shall prove that Prawitz candidates are an interpretation of the quantifier-free fragment of C_Σ.

Definition 6.2 *Let* $C \in \mathbf{GC}$.
(i) C *is a* Prawitz candidate *if*

$$(\forall q = in\neg C\ p[x := q] \in SN_\perp) \Rightarrow \mathcal{N}x.p \in C$$

(ii) $\mathbf{PC} = \{C \in \mathbf{GC} | C$ *is a Prawitz candidate* $\}$.

The Prawitz candidates soundly interpret the double negation rules by definition. Lemmas 6.1(ii),6.2,6.4 still hold for $\mathbf{PC}$ because $\mathbf{PC} \subseteq \mathbf{GC}$. Thus Prawitz candidates soundly interpret all propositional rules. In order to have an interpretation of the quantifier-free fragment of C_Σ we have only to check that $SN_A \in \mathbf{PC}$ for A atomic and that $\mathbf{PC}$ soundly interpret the connectives $\neg$ and $\wedge$, i.e. that $\mathbf{PC}$ is closed under $\underline{\neg}$ and $\underline{\triangle}$.

Lemma 6.5 *Let A be an atomic proposition. Then* $SN_A \in \mathbf{PC}$.

Proof. $SN_A \in \mathbf{GC}$ by Lemma 6.1(i)??. Thus we have only to check that SN_A is a Prawitz candidate, i.e. that

$$(\forall q \in \neg SN_A\ p[x := q] \in SN_\perp) \Rightarrow \mathcal{N}x.p \in SN_A.$$

Since $x^{\neg A} \in \neg SN_A$, then $p \equiv p[x : x^{\neg A}] \in SN_\perp$. Let n be a bound for p. We prove $\mathcal{N}x.p \in SN_A$ by induction on n. By (cand 3) for SN_A, it is enough to prove that for all proofs q, $(\mathcal{N}x.p \longrightarrow q) \Rightarrow q \in SN_A$. If the reduction takes places in p, then q is $\mathcal{N}x.p'$ for some p' such that $p \longrightarrow p'$. In such a case, by the induction hypothesis and the fact that $p[x := q] \longrightarrow p'[x := q]$, we conclude that $q \equiv \mathcal{N}x.p' \in SN_A$. If $(\mathcal{N}x.p)^A$ is the redex being reduced then, since A is atomic, we must have applied a reduction (At-class). Thus q is a subproof of p and, since $p \in SN_\perp$, $q \in SN_A$. □

We have now to prove that $\mathbf{PC}$ soundly interpret the formation of quantifier-free proposition, i.e. that $\mathbf{PC}$ is closed under $\underline{\neg}$ and $\underline{\triangle}$. To make such proof easier we first prove the following lemma.

Lemma 6.6 *Let* $C_1, C_2, C \in \mathbf{GC}$ *and* $C_i^i = A_i, C^* = A$.
(i) $p^{\neg A_i} \in \neg C_i \quad \Rightarrow \quad \sigma_i[p] \in \underline{\neg}(C_1 \underline{\triangle} C_2)$ *i=1,2*
(ii) $p^A \in C \quad \Rightarrow \quad \sigma[p] \in \underline{\neg}\underline{\neg} C$

Proof. (i) Let $p \in \underline{\neg} C_i$. By definition, $\sigma_i[p] \equiv \lambda x^{A_1 \wedge A_2}.p^{A_i}(\pi_i(x)^{A_i})^\perp$. By Lemma 6.4(iii), in order to have $\sigma_i[p] \in \neg(C_1 \underline{\triangle} C_2)$ it is enough to show that $\forall q \in C_1 \underline{\triangle} C_2\ p(\pi_i(q)) \in SN_\perp$. This follows from Lemma 6.4(ii)(iv). □

Lemma 6.7 *(i)* $\underline{\triangle} : \mathbf{PC} \to \mathbf{PC} \ \to \mathbf{PC}$
(ii) $\underline{\triangle} : \mathbf{PC} \to \mathbf{PC}$.

Proof. Let $C_1, C_2, C \in \mathbf{PC}$.
(i) By Lemma 6.3(i) $C_1 \triangle C_2 \in \mathbf{GC}$. We have still to check that $C_1 \triangle C_2$ is a Prawitz candidate, i.e. that $(\forall q \in \neg(C_1 \triangle C_2)\ p[x := q] \in SN_\perp) \Rightarrow \mathcal{N}x.p \in C_1 \triangle C_2$.
As we did in the proof of Lemma 6.5, we first remark that $p \in SN_\perp$, and then we proceed by induction on the bound n of p. By (cand 3) we have to prove that for all proofs q, $(\mathcal{N}x.p \longrightarrow q) \Rightarrow q \in C_1 \triangle C_2$. The only interesting case is when $(\mathcal{N}x.p)^{A_1 \wedge A_2}$ is the redex being reduced. In such a case we must perform a reduction ($\wedge$-class), i.e.

$$\mathcal{N}x.p \longrightarrow q \equiv < \mathcal{N}x_1.p[\sigma_1[x]], \mathcal{N}x_2.p[\sigma_2[x]] >$$

and we have to prove that $q \in C_1 \triangle C_2$.
By Lemma 6.4(i), it is enough to prove that $\mathcal{N}x_i.p[\sigma_i[x]] \in C_i$ for $i = 1, 2$. Since $C_i \in \mathbf{PC}$, we have in fact to prove $\forall q \in \underline{\neg} C_i\ p[\sigma_i[q]] \in SN_\perp$.
By Lemma 6.6(i), $\sigma_i[q] \in \underline{\neg}(C_1 \triangle C_2)$. Thus, by hypothesis on $p, p[\sigma_i[q]] \in SN_\perp$.
(ii) By Lemma 6.3(ii), $\underline{\neg} C \in \mathbf{GC}$. We have still to check that $\underline{\neg} C$ is a Prawitz candidate, i.e. that $(\forall q \in \underline{\neg}\underline{\neg} C\ p[x := q] \in SN_\perp) \Rightarrow \mathcal{N}x.p \in \underline{\neg} C$.
As we did in the proof of Lemma 6.5, we first remark that $p \in SN_\perp$, and then we proceed by induction on the bound n of p. By (cand 3) it is enough to prove that for all proofs q, $(\mathcal{N}x.p \longrightarrow q) \Rightarrow q \in \underline{\neg} C$. The only interesting case is when $(\mathcal{N}x.p)^{\neg A}$ is the redex being reduced. In such a case we must perform a reduction ($\neg$-class), i.e. $\mathcal{N}x.p[x] \longrightarrow q \equiv \lambda x_1.p[\sigma[x_1]]$.
By Lemma 6.4(i), it is enough to prove that $\forall a \in C\ p[\sigma[a]] \in SN_\perp$. By Lemma 6.6(ii), $\sigma[a] \in \underline{\neg}\underline{\neg} C$. Thus, by hypothesis on p, $p[\sigma[a]] \in SN_\perp$. □

In the next section we shall extend the interpretation given here to the whole system C_Σ.

6.3 Predicative and Impredicative Quantification

In this subsection we interpret the types of C_Σ as particular sets, and quantification as an operation on such sets. Then we shall prove that such an interpretation is sound with respect to the formation of quantified propositions and with respect to the rules for such propositions.

For each type T of C_Σ we define a set $[\![T]\!]$ of higher order candidates, and a $\beta\eta$-normal index $P : T$ for each $F \in [\![T]\!]$. The definition is by simultaneous induction on T.

Definition 6.3 – $[\![Prop]\!] = \mathbf{PC}$
– $[\![\iota]\!] = \{$ *the terms of* $\mathsf{C}_\Sigma\}$
If $C \in \mathbf{PC}$ *the only index of* C *is* C^*.
If $t \in [\![\iota]\!]$ *the only index of* t *is* t *itself.*
– $[\![T \to U]\!] = \{F : [\![T]\!] \to [\![U]\!] | F$ *has some index* $\}$.
where $F : [\![T]\!] \to [\![U]\!]$ *has index* P *if* $\forall a \in [\![T]\!] (a$ *has index* $E) \Rightarrow F(a)$ *has index (the* $\beta\eta$ *normal form of)* $P(E)$.

We define now some canonical elements in each $[\![T]\!]$.

Definition 6.4 *(i) Let* t *be a term. Then* $SN_t =_{def} t \in [\![\iota]\!]$.
(ii) Let P *be an atomic proposition. Then* SN_P *has been already defined at the beginning of 6.1*
(iii) Let P *be any atomic predicate of type* $T \to U$ *(i.e. an expression of the form* $X(E_1, \dots, E_n)$*) and* $G \in [\![T]\!]$ *with index* Q*. Then we put*

$$SN_P(G) =_{def} SN_{P(Q)} \in [\![U]\!]$$

Lemma 6.8 *Let T be a type and $E : T$.*
(i) Each $F \in [\![T]\!]$ has a unique ($\beta\eta$-normal) index
(ii) SN_E is a well defined element of $[\![T]\!]$ with index E.

Proof. By simultaneous induction on T.
The cases $T = Prop, \iota$ are trivial or already proved.
Let $T = U \to V$. Suppose $F \in [\![T]\!]$ have indexes P_1, P_2. By induction hypothesis, if X is a variable and $X : U$ then $SN_X \in [\![U]\!]$ and SN_X has index X. Thus $F(SN_X)$ has indexes $P_1(X)$ and $P_2(X)$. By induction hypothesis on U, $P_1(X) =_\beta P_2(X)$ and we deduce $P_1 =_{\beta\eta} P_2$ by η. Since P_1, P_2 are $\beta\eta$-normal, $P_1 \equiv P_2$. We want to prove now that SN_E is a well defined element of $[\![T]\!]$ with index E. We have only to check that, if $G \in [\![U]\!]$ has index Q, then $SN_E(G)(\equiv SN_{E(Q)})$ is an uniquely determined element of $[\![U]\!]$ with index $E(Q)$. By the induction hypothesis on U, the index of G is unique; thus $SN_{E(Q)}$ is uniquely determined and , by the induction hypothesis on U, it is in $[\![U]\!]$. $\Box$

If T is a type and $F \in [\![T]\!]$, we shall denote by F^* (expression of type T) the only index of F. Remark that, by definition of $[\![T]\!]$, if $T = U \to V$, we have that $F(G)^* =_{\beta\eta} F^*(G^*)$.

For each type T we define now a higher order candidate

$$\underline{\forall} \in [\![(T \to Prop) \to Prop]\!]$$

with index $\lambda P.\forall X.P(X)$, to interpret quantification. We shall drop the subscript T when it is irrelevant.

Definition 6.5 *Let $F \in [\![T \to Prop]\!]$.*

$$\underline{\forall}_T(F) =_{def} \{p \ proof \ | \forall G \in [\![T]\!] p(G^*) \in F(A)\}$$

By type consistency, each $p \in \underline{\forall}_T(F)$ has type $\forall x^T.F^*(x)$.
To prove that $\underline{\forall}_T \in [\![(T \to Prop) \to Prop]\!]$ it is enough to check that $\underline{\forall}(P) \in [\![Prop]\!]$. We do it in two steps; first we check that $\underline{\forall}(F) \in \mathbf{GC}$, and then that $\underline{\forall}(F) \in \mathbf{PC}$.

Lemma 6.9 *Let $F \in [\![T \to Prop]\!]$. Then $\underline{\forall}(F) \in \mathbf{GC}$.*

Proof. We check (cand 1), (cand 2) and (cand 3).
(cand 1) Let $p \in \underline{\forall}(F)$. If X is a variable and $X : T$ then $SN_X \in [\![T]\!]$ and therefore $p(X) \in F(SN_X) \subseteq SN_{F^*(X)}$. Thus p strongly normalizes.
(cand 2) Let $p \in \underline{\forall}(F)$ and $p \longrightarrow q$. Then $\forall G \in [\![T]\!] p(G^*) \in F(G)$ and $p(G^*) \longrightarrow q(G^*)$ Thus $\forall G \in [\![T]\!] q(G^*) \in F(G)$ and we conclude that $q \in \underline{\forall}(F)$.
(cand 3) Suppose that for all proofs q, $(p \longrightarrow q) \Rightarrow q \in \underline{\forall}(F)$ and that p does not begin with an introduction. We want to prove $\underline{\forall} \in [\![T]\!] p(G^*) \in F(G)$. We first remark that, by (cand 3) for $SN_{\forall X.F^*(X)}$, since for all proofs q, $(p \longrightarrow q) \Rightarrow q \in \underline{\forall}(F) \subseteq SN_{\forall X.F^*(X)}$, we can deduce $p \in SN_{\forall X.F^*(X)}$. By (cand 3) for $F(G)$ it is enough to prove that for all proofs q', $(p(G^*) \longrightarrow q') \Rightarrow q' \in F(G)$. Since p does not begin with a λ, the reduction on $p(G^*)$ either can take place in p or can be a reduction (At-class) with redex $p(G^*$ with $F(G^*)$ atomic. In the first case q' is $q(G^*)$ for some q such that $p \longrightarrow q$ and then we have $q \in \underline{\forall}(F)$ by hypothesis, and therefore $q(G^*) \in F(G)$. In the second case q' is a subproof of $p \in SN_{\forall X.F^*(X)}$ and therefore $q' \in SN_{F(a)^*}$. By Lemma 6.1(ii), since $F(G)^*$ is atomic, we have that $F(a) = SN_{F(a)^*}$ and we can conclude that $q \in F(a)$. $\Box$

The lemma above then states that $\underline{\forall}$ soundly interpret the *intuitionistic* quantification.

We prove now that the intuitionistic rules for quantification have a sound interpretation.

Lemma 6.10 *Let T be a type and $F \in [\![T \to Prop]\!]$.*
(i) $(\forall a \in [\![T]\!]\ p[x := G^*] \in F(G)) \Rightarrow \lambda x.p \in \underline{\forall}(F)$
(ii) $p \in \underline{\forall}(F), G \in [\![T]\!] \Rightarrow p(G^*) \in F(G)$.

Proof. (i) We have to prove that $\forall G \in [\![T]\!]\ (\lambda x.p)G^* \in F(G)$. By (cand 3), since such proof begins with an application, it is enough to prove that for all proofs q, $((\lambda x.p) \longrightarrow q) \Rightarrow q \in F(G)$. We first remark that if X is a variable and $X : T$ then $SN_X \in [\![T]\!]$ and therefore $p \equiv p[x := x] \in F(SN_X) \subseteq SN_{F^*(X)}$. Let n be a bound for p. We prove $(\lambda x.p)G^* \in F(G)$ by induction on n. If $q \equiv (\lambda x.p')G^*$ with $p \longrightarrow p'$, then we still have $p'[x := G^*] \in F(G)$ by (cand 2) and $p[x := G^*] \longrightarrow p'[x := G^*]$. Thus, by the induction hypothesis we have $q \equiv (\lambda x.p)G^* \in F(G)$. If $(\lambda x.p)G^*$ is the redex being reduced, then either $(\lambda x.p)G^* \longrightarrow p[x := G^*] \equiv q$ or we perform a reduction (At-class), $F(G)^*$ is atomic and q is a subproof of p. In the first case, $q \in F(G)$ by hypothesis on p. In the seco-nd case $q \in SN_{F(G)^*}$ because p strongly normalizes. Since $F(a)$ is atomic, by Lemma 6.1(ii) $F(G) = SN_{F(G)^*}$ and we conclude that $q \in F(G)$. □

We prove now that $\underline{\forall}$ soundly interpret also the *classic* quantification, i.e. that $\underline{\forall}(F)$ is also a Prawitz candidate. In order to make the proof easier, we first prove the following lemma.

Lemma 6.11 *Let T be a type and $F \in [\![T \to Prop]\!]$. Then, for all $G \in [\![U]\!]$*

$$q \in \underline{\neg}F(G) \Rightarrow \sigma_{G^*}[q] \in \underline{\neg\forall}(F).$$

Proof. We know that $\sigma_{G^*}[q] = \lambda x.q(x(G^*))$. By Lemma 6.4(iii) $\sigma_{G^*}[q] \in \underline{\neg\forall}(F)$ if for all $q' \in \underline{\forall}(F)$ we have that $q(q'(G^*)) \in SN_\perp$. This fact follows by definition of $\underline{\forall}(F)$ and $\underline{\neg}F(G)$. □

Lemma 6.12

$$\underline{\forall}_T \in [\![(T \to Prop) \to Prop]\!]$$

Proof. Let $F \in [\![T \to Prop]\!]$. We proved in Lemma 7.9 that $\underline{\forall}(F) \in \mathbf{GC}$. There is only left to prove

$$(\forall q \in \neg\underline{\forall}(F) p[x := q] \in SN_\perp) \Rightarrow \mathcal{N}x.p \in \underline{\forall}(F).$$

Let $x : \neg\forall z.F^*(z)$. Then $x \in \underline{\neg}(F)$ and therefore $p \equiv p[x := x] \in SN_\perp$. Let n be a bound for p. We prove $\mathcal{N}x.p \in \underline{\forall}(F)$ by induction on n. By (cand 3) for $\underline{\forall}(F)$ it is enough to prove that for all q' $(\mathcal{N}x.p \longrightarrow q') \Rightarrow q' \in \underline{\forall}(F)$. If q' is $\mathcal{N}x.p$ with $p \longrightarrow p'$ then we still have $\forall q \in \underline{\neg\forall}(F)$ $p'[x := q] \in SN_\perp$ because $p[x := q] \longrightarrow p'[x := q] \in SN_\perp$. Since p' has a smaller bound than p, by the induction hypothesis we have $q' \equiv \mathcal{N}x.p' \in \underline{\forall}(F)$. If $\mathcal{N}x.p$ is the redex being reduced, then we must apply a reduction ($\forall$-class) and q' is $\lambda z.\mathcal{N}x_1 p[\sigma_z[x_1]]$. By Lemma 6.10(i), to prove $q' \in \underline{\forall}(F)$ it is enough to prove $\forall G \in [\![T]\!]\ \mathcal{N}x_1.p[\sigma_{G^*}[x_1]] \in F(G)$. Since $F(G) \in [\![Prop]\!] \equiv \mathbf{PC}$, we are reduced to prove $\forall q \in \underline{\neg}F(G)\ p[\sigma_{G^*}[q]] \in SN_\perp$.
By Lemma 6.11 $q \in \underline{\neg}F(G) \Rightarrow \sigma_{G^*}[q] \in \underline{\neg\forall}(F)$. Thus, by hypothesis on p, we conclude that $p[\sigma_{G^*} \in SN_\perp$. □

In the next subsection we shall collect the work done up to now and shall define a formal Realization semantics for C_Σ. Strong normalization for proofs in C_Σ will follow as corollary.

6.4 A Realization Semantic for C_Σ

In this subsection we shall formally define a Realization Semantics for C_Σ, associating to each proposition A a "natural" candidate C for A. Then we shall prove that such Realization Semantics is sound for C_Σ, i.e. that

$$\mathbf{x} : \mathbf{A} \vdash_{\mathsf{C}_\Sigma} \Rightarrow p \in C.$$

Since $C \subseteq SN_A$, it will follow that each proof p of C_Σ strongly normalizes.

We first interpret, inside the Realization Semantics, predicates and terms. In the previous subsections we have shown how to interpret each symbol inside a predicate but variables. An interpretation for variables is provided by the notion of *predicate assignment*.

Definition 6.6 *A Predicate assignment ρ is a function with a finite set of variables $\{v_1, \ldots, v_2\}$ as domain, such that, for each $v \in dom(\rho)$, each type T and each proposition A we have :*
(i) $v : T \Rightarrow \rho(v) \in [\![T]\!]$
(ii) $v : A \Rightarrow \rho(v) : A$

Each variable v is interpreted, in a given assignment ρ, by $\rho(v)$. $\rho(v)$ is a higher order predicate if v is a predicate or term variable, and it is a proof if v is a proof variable. We can associate a substitution ρ^* to ρ in the following way.

Definition 6.7 *Let x be any variable, T a type and A a proposition.*
(i) $dom(\rho)^* =_{def} dom(\rho))$
(ii) $x \in dom(\rho),\ x : T \Rightarrow \rho^*(x) =_{def} \rho(x)^*$
(iii) $x \in dom(\rho),\ x : A \Rightarrow \rho^*(x) =_{def} \rho(x)$

From now on, if $x \notin dom(\rho), x : T, G \in [\![T]\!]$, we shall denote by $\rho_{x,G}$ the extension of ρ defined by

$$\rho_{x,G}(x) =_{def} G.$$

$\rho_{x,G}$ is still a predicate assignment.

We are now ready to interpret any predicate of C_Σ.

Definition 6.8 *Given a predicate assignment ρ, we extend it to a predicate $P : T$ such that $FV(P) \subseteq dom(\rho)$ by induction on P, as follows :*
$\rho(At) = SN_{At}$ *if At is a proposition of Σ*
$\rho(A \wedge B) = \rho(A) \underline{\wedge} \rho(B)$
$\rho(\neg A) = \underline{\neg} \rho(A)$
$\rho(\lambda x^T.P)$ *defined by* $\rho(\lambda x^T.P)(G) = \rho_{x,G}(P), \forall G \in [\![T]\!]$
$\rho(PQ) = \rho(P)\rho(Q)$
$\rho(\forall x.A) = \underline{\forall}(\rho(\lambda x.A))$

Lemma 6.13 *Let ρ, P, T be as above. Then $\rho(P)$ is a well-defined element of $[\![T]\!]$ with index $\rho(P)^* = \rho^*(P)$.*

Proof. By induction on P, distinguishing the cases according to the outermost symbol of P.

- $P = At$. Then $\rho(At) = SN_{At} \in [\![Prop]\!] = \mathbf{PC}$ by Lemma 6.5. Moreover $SN_{At}^* = At$.

- $P = A \wedge B$. Then $\rho(A \wedge B) = \rho(A) \underline{\wedge} \rho(B) \in [\![Prop]\!] = \mathbf{PC}$ by Lemma 6.7 and $\rho(A), \rho(B) \in [\![Prop]\!] = \mathbf{PC}$. Moreover $\rho(A \wedge B)^* = (\rho(A) \underline{\wedge} \rho(B))^* = \rho(A)^* \wedge \rho(B)^* =$ (by the induction hypothesis) $\rho^*(A) \wedge \rho^*(B) = \rho^*(A \wedge B)$.

- $P = \neg A$. Then $\rho(\neg A) = \underline{\neg} \rho(A) \in [\![Prop]\!] = \mathbf{PC}$ by Lemma 6.7. Moreover $\rho(\neg A)^* = (\underline{\neg} \rho(A))^* = \neg \rho(A)^* =$ (by the induction hypothesis) $\neg \rho^*(A) = \rho^*(\neg A)$.

- $P = \lambda x^U.Q^V$. Then if $G \in [\![U]\!]$ then $\rho(\lambda x.Q)(G) = \rho_{x,G}(Q) \in [\![U]\!]$ by the induction hypothesis on Q. Thus $\rho(\lambda x.Q) : [\![U]\!] \rightarrow [\![V]\!]$. Moreover $\rho(\lambda x.Q)(G)^* = \rho_{x,G}(Q)^* =$(by the induction hypothesis) $\rho^*_{x,G}(Q) = (\rho^*, x := G^*)(Q) = \rho^*(Q)[x := G^*] =_\beta (\lambda x.\rho^*(Q))(G^*) = \rho^*(\lambda x.Q)(G^*)$. Thus $\rho(\lambda x.Q)$ has index $\rho^*(\lambda x.Q)$.

- $P = Q^{U\to V}R^U$. Then $\rho(QR) = \rho(Q)(\rho(R)) \in [\![V]\!]$ because $\rho(Q) \in [\![U \to V]\!]$ and $\rho(R) \in [\![u]\!]$. Moreover $\rho(QR)^* = (\rho(Q)\rho(R))^* = \rho(Q)^*\rho(R)^* =$ (by the induction hypothesis) $\rho^*(Q)\rho^*(R) = \rho^*(QR)$.
- $P = \forall x^T.A$. Then $\rho(\forall x^T.A) = \underline{\forall}(\rho(\lambda x^T.A)) \in [\![Prop]\!]$ because $\underline{\forall} \in [\![(T \to Prop) \to Prop]\!]$ by Lemma 6.12 and $\rho(\lambda x^T.A) \in [\![T \to Prop]\!]$. Moreover $\rho(\forall x.A)^* = \forall(\rho(\lambda x.A))^* = \forall x.\rho(\lambda x.A)^*(x) =$ (by the induction hypothesis) $\forall x.\rho^*(\lambda x.A)(x) = \forall x.\lambda x.\rho^*(A)(x) = \forall x.\rho^*(A) = \rho^*(\forall x.A)$.

□

In order to state a soundness theorem for the interpretation $\rho(P)$ we have just outlined, we need one more definition.

Definition 6.9 *An assignment ρ is a predicate assignment such that, for each variable $x \in dom(\rho)$ and proposition A we have*

$$x : A \Rightarrow \rho(x) \in \rho(A).$$

Theorem 6.1 (Soundness Theorem) *Let ρ be an assignment and p be a proof such that $FV(p) \subseteq dom(\rho)$. Then*

$$p : A \Rightarrow \rho^*(p) \in \rho(A).$$

Proof. By induction on p, distinguishing the cases according to the outermost symbol of p.

- $p = x^A$. By definition of assignment and $\rho^*(x) = \rho(x)$.
- $p = r(\mathbf{t}, p_1, \ldots, p_n)$. By the induction hypothesis and lemma 6.2.
- $p = < p_1, p_2 >$. By the induction hypothesis and Lemma 6.4(i).
- $p = \pi_i(q)$. By the induction hypothesis and Lemma 6.4(ii).
- $p = (\lambda x^B.q^\perp)^{\neg B}$. By the induction hypothesis and Lemma 6.4(iii).
- $p = (p_1^{\neg B}p_2^B)^\perp$. By the induction hypothesis and Lemma 6.4(iv).
- $p = (\lambda x^T.q^A)^{\forall x^T.A}$. By the induction hypothesis and Lemma 6.10(i).
- $p = (q^{\forall x.A}(E))^{A[x:=E]}$. By the induction hypothesis and Lemma 6.10(ii) we may deduce $p \in \rho(\lambda x.A)(\rho(E)) = \rho_{x,\rho(E)}(A)$. The thesis follows from the **claim** :

$$\rho_{x,\rho(E)}(A) = \rho(A[x := E]).$$

- $p = (\mathcal{N}x^{\neg B}q^\perp)^B$. By the induction hypothesis and the definition of Prawitz candidate.

□

Corollary *Each proof of C_Σ strongly normalizes.*
Proof. Let $p : A, A : Prop$ and consider the assignment ρ defined by :
if $x : T, x \in FV(p)$ then $\rho(x) =_{def} SN_x$
if $x : B, x \in FV(p)$ then $\rho(x) =_{def} x$.
Then $\rho^*(x) = x$ for each $x \in FV(p)$ and $\rho^*(p) = p$. By the previous theorem $p = \rho^*(p) \in \rho(A) \subseteq SN_{\rho^*(A)} = SN_A$. Thus, each proof p strongly normalizes. □

There is still left to prove the claim $\rho_{x,\rho(E)}(A) = \rho(A[x := E])$. Recall that $A[x := E]$ denote the $\beta\eta$-normal form of A with E replaced for x. To consider the substitution alone we give the following definition.

Definition 6.10 *$sub(A, x, E)$ = the result of replacing each free occurrence of x by E in A.*

With the definition above $A[x := E]$ is the $\beta\eta$-normal form of $sub(A, x, E)$. We can now split the proof of the claim in two parts : first we prove that $\rho(sub(A, x, E)) = \rho_{x,\rho(E)}$ and second we can prove that $A =_{\beta\eta} B \Rightarrow \rho(A) = \rho(B)$. In both cases we use the following property.

Lemma 6.14 *Let P be a predicate ρ_1, ρ_2 be an assignment and suppose that $\rho_1(x) = \rho_2(x)$ for each $x \in FV(p)$. Then $\rho_1(p) = \rho_2(p)$.*

Proof. By induction on p. We might also remark that we never use in the definition of p the values $\rho_1(x), \rho_2(x)$ for $x \notin FV(p)$. □

We can now prove the following lemma.

Lemma 6.15 *Let P be a predicate, E an expression, x a variable with the type of E , and ρ an assignment such that $FV(P), FV(E) \subseteq dom(\rho)$. Then*

$$\rho(sub(P, x, E)) = \rho_{x,\rho(E)}(P).$$

Proof. By induction on P. The only non-trivial case is when $P = \lambda y^T.Q$. In such a case we may suppose that $x \neq y$ and have to prove $\rho(sub(\lambda y^T.Q, x, E)) = \rho(\lambda y^T.sub(Q, x, E))$. We have to check that, for every $G \in [\![T]\!]$

$$\rho_{y,G}(sub(Q, x, E)) = (\rho_{x,\rho(E)})_{y,G}(Q).$$

By the induction hypothesis on Q we have that

$$\rho_{y,G}(sub(Q, x, E)) = (\rho_{y,G})_{x,\rho_{y,G}(E)}(Q).$$

By Lemma 6.14 $\rho_{y,G}(E) = \rho(E)$, thus

$$(\rho_{y,G})_{x,\rho_{y,G}(E)}(Q) = (\rho_{y,G})_{x,\rho(E)}(Q) = (\rho_{x,\rho(E)})_{y,G}(Q).$$

□

We can now prove the claim by proving

$$\rho(sub(A, x, E)) = \rho(A[x := E]).$$

This follows from the following lemma.

Lemma 6.16 *Let P, G be predicates, $P =_{\beta\eta} G$, ρ be an assignment and $FV(P), FV(Q) \subseteq dom(\rho)$. Then*

$$\rho(P) = \rho(Q).$$

Proof. It is enough to prove that, if $P \longrightarrow_{\beta\eta} P'$, then $\rho(P) = \rho(P')$. this fact can be proved by induction on P. The only non-trivial case is when P itself is the redex being reduced, i.e. when $P \equiv (\lambda x.Q)R \longrightarrow sub(Q, x, R)$ or $P \equiv \lambda x.Q(x) \longrightarrow Q$(with $x \notin FV(Q)$). In the first case we apply Lemma 6.15, in the second one Lemma 6.14 and the fact that $x \notin FV(Q)$. □

References

1. Clinger W., Rees J. The *revised*^{3} report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37-79, 1986.

2. Felleisen M., Friedman D., Kohlbecker E., Duba B. Reasoning with continuations. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, 131-141, 1986.

3. Friedman H. Classically and intuitionistically provably recursive functions. In Scott D.S. and Muller G.H. editors, *Higher Set Theory*, vol.699 of *Lecture Notes in Mathematics*, 21-28. Springer Verlag, 1978.

4. Girard J.-Y. Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *Proceedings of the second Scandivian Logic Symposium*, eds.J.E. Fenstad, Amsterdam, North-Holland, 63-92, 1971.

5. Griffin T.G. A formulae-as -types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

6. Kreisel G. Mathematical significance of consistency proofs. *Journal of Symbolic Logic*, 23:155-182, 1958.

7. Murthy C. Extracting constructive content from classical proofs. Ph.d. thesis, Department of Computer science, Cornell University, 1990.

8. Plotkin G. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 125-159, 1975.

9. Prawitz D. Natural deduction, a proof theoretical study. Stockolm, Almqvist and Winskell, 1965.

10. Prawitz D. Validity and normalizability of proofs in 1-st and 2-nd order classical and intuitionistic logic. In *Atti del I Congresso Italiano di Logica*, Napoli, Bibliopolis, 11-36, 1981.

Metalogical Frameworks

David A. Basin[*] Robert L. Constable[†]

July 1991

Abstract

In computer science we speak of *implementing* a logic; this is done in a programming language, such as Lisp, called here the *implementation language*. We also reason about the logic, as in understanding how to search for proofs or in knowing it is consistent; in the terminology of mathematical logic, these arguments are conducted in the *metalanguage* of the *object language* being implemented. We also reason about the implementation itself, say to know it is correct. This is done in a *programming logic*. How do all these logics relate? This paper considers that question and more.

We show that by taking the view that the metalogic is primary, these other parts are related in standard ways. The metalogic must be suitably rich so that the object logic can be presented as an abstract data type, and it must be suitably computational (or constructive) so that an instance of that type is an implementation. The data type abstractly encodes all that is relevant for metareasoning, i.e. not only the term constructing functions but also the principles for reasoning about arbitrary terms and computing with them.

Our work can also be seen as an approach to the task of finding a generic way to present logics and their implementations, which is for example the goal of the Edinburgh Logical Frameworks (ELF) effort. This approach extends well beyond proof-construction and includes computational metatheory as well.

1 Introduction

1.1 Role of Logical Frameworks and Formalized Metamathematics

At one time logic seemed to be a *finished* subject, and computers, like radio telescopes and linear accelerators, seemed to be the tools of *big science*. Now computers are ubiquitous and their software systems have brought logics to life. We see limitless expansion in both domains with computers destined to play a part in nearly every aspect of life and the logics of their systems needed to bring order and sense to this activity, as well as provide a basis for enabling the realization of ever bolder dreams. This is a new role for logic and with it come a host of new tasks and problems; this paper is about some of them, especially those that fall to computer scientists to address. Part of the job can be understood in terms of past accomplishments in computer science, and part belongs to a new subfield. Let us be more specific about this new role for logic and put these tasks in context, starting with the most familiar.

[*]Department of Artificial Intelligence, University of Edinburgh, Scotland. Email: basin@aipna.ed.ac.uk. Partially supported by the North Atlantic Treaty Organization under a grant awarded in 1990.

[†]Department of Computer Science, Cornell University, Ithaca NY, USA. Email: rc@cs.cornell.edu. Partially supported by an SERC visiting fellowship and a Guggenheim fellowship.

LF in General Sense

- Wide variety of logics **demands** general tools ("Next 700 Theorem Provers")

- How to provide generic tools for computer implementations
  - define the logic
    - syntax, proof rules
  - automate theorem proving
    - proof search

Formalized Metamathematics

- Mechanized reasoning **demands** general metareasoning facilities ("Metalogical Frameworks")

- How to formalize metamathematics of logics
  - represent the logic
    - wffs, proofs, semantics
  - metatheory about proofs
    - derived rules
    - decision procedures
    - normal form theorems

Figure 1: Congruence of Concerns

The first generation of software systems were brought under control with the development of automata theory, formal languages, and programming logics. Modern compilers are built with tools and techniques based on a deep understanding of parsing and compilation, and the methods of rigorous program development make it possible to achieve high levels of reliability for a range of specifiable programs.

Now the field seeks to do more, to extend the previous successes to a wider range of computing problems and to carry the techniques forward into more imaginative experimental systems which have formal logics as components. For instance there will be programming languages with type systems that are equivalent to logics, and the type checkers will be little theorem provers (maybe even big ones). There will be formal specification languages and program verifiers both based on a formal logic of some kind. There will be systems that support program synthesis and hardware synthesis. There will be data bases with rich query languages that are formal logics. There will be more experimental systems such as *problem solving environments* and programmer's assistants. In all of these systems formal logic is playing its new role. Moreover, there will be many different logics and many different implementations of each. So as in the case of compiler construction, there will be an industry for building and modifying the implemented logics.

Thus far there are few tools designed specifically to support the activity of implementing logics. So systems are built from scratch using tools intended for other purposes. What is wanting is a framework for designing these logics and supplying the generic tools. One of the main goals of research in *logical frameworks* is to offer a setting to guide the building of these generic tools.

Another way to approach the topic of this paper is to consider the activity of generating *formal mathematics*, the project of Automath [8]. This has been taken up with considerable energy in computer science because it is seen as an integral part of some of the new kinds of systems mentioned above and because it is related to concerns of AI and because it is a generalization of programming, an activity central to computer science. For whatever reason (and there maybe other more subtle ones), this is now done by computer scientists. One of the subjects that is most promising to formalize is metamathematics, in part because the results of that work feed back directly to the task of generating formal theorems. In the context of formalizing metamathematics, we are confronted with many of the same questions facing the designer of a logical framework; some of these parallels are drawn in Figure 1. This paper considers why this is so. Moreover we claim that the vantage point of formalized metamathematics is a good way to look at the task of logical frameworks, and we advocate its primacy, hence the title of the paper.

Scientists actually doing formal mathematics have come to understand the importance of metareasoning from experience. They quickly saw that nearly all informal mathematics is a mixture of object theory and metatheory. The work of Howe [25], comes to our minds in this connection. Also the style of automating reasoning via tactics involves programming in a *metalanguage*, and if one applies the methodology of programming systems based on the proofs-as-programs principle, such as Nuprl [12] and the Calculus of Constructions (CoC) [14], then many of these tactics should be extracted from metatheorems. A great deal of work has been done in this direction at Cornell, [27, 24, 2]. Also even if this principle is not underlying the programming style, it has been widely noticed that substantial economies in proof generation are possible if tactics are replaced by the metatheorems that they implement.

The key component, and least understood one, in an implemented logic is usually the *inference engine*, and our starting point is the observation that the experimental systems will need ways to quickly modify them, even the workhorse system will need ways to progressively enhance it. Our response to the situation is to conjecture that the problems of understanding, specifying and modifying the inference engine are best dealt with in the framework of a *formalized metalogic*, not in the setting of a hybrid system of languages nor in a framework that enables a generic definition of the syntax and proof structure of a logic. It will be important to reason about the object logic (the one being implemented) in a very expressive metalogic.

1.2 Conceptual Issues

Logicians have been concerned for years with treating their subject more abstractly and generally. The elementary formal systems of Smullyan were an attempt to characterize the deductive machinery of *Principia Mathematica*, and lately Feferman has proposed more usable formalisms [19, 18] such as FS_0 for expressing proof theory. There are abstract semantic characterizations of logic [4, 5] and the theorem of Lindström in abstract model theory characterizing first-order logics. There are other efforts along these lines such as [1].

The work of logicians deals with many of the issues central to the problem of logical frameworks such as what is a logic, what are the different ways to present them and which are essentially the same, which are different. Their concern for the most general setting for major results such as Godel's theorems, Löb's theorem, Girard's normalization methods, etc. is directly germane to the task of formalizing metamathematics. But some of the issues faced in defining a logical framework and formalizing mathematics on a machine are new. For example, how should bound variables be represented, as deBruijn indices or as special terms? How can a sequent based proof economize on storage? What is the best way to realize computational content? What is the essential difference between sequent style proofs and natural deduction style, and does it apply to all logics?

Inevitably discoveries about logical frameworks will contribute to a conceptual understanding of logic. From the vantage point of this paper we see that the efforts to formalize metamathematics will contribute as well. This might not be so obvious *a priori*.

1.3 Results

Beyond advocating the primacy of metalogic this paper contributes some specific technical results. We propose a particular way to specify logics, namely using types that one might call *higher-order abstract data types* or ADT's for this paper. We show that an implementation of a logic can be seen then as an instance of the abstract data type. There are many interesting connections between these types and other approaches to data abstraction in type theory and programming languages.

We also look at *constructive metatheory in action* and illustrate by example why a framework for automating logic will want access to the results of formalized metamathematics. This practical need is remarkably congruent to the philosophical stance adopted by the first metamathematicians who used finitist or constructivist metatheories. This historical accident means that there is a deep literature of informal implicitly computational metatheory to draw on as well the explicitly computational results of modern computer scientists (unification, resolution, term rewriting, etc.) such as decision procedures, and moreover that formalizing these results makes them clearer and more general, thus fulfilling the promise of formalized mathematics. Indeed given the deep literature in proof theory, there is much to be done in this vein to bring it to life and make it available to applied logicians. But also new kinds of metamathematical result are needed in reasoning about implemented logics, and we mention some of them as well.

Finally we show that by taking a rich constructive metalogic as the basis for a logical framework, the context becomes simpler and the relationship between the concerns of logical frameworks and those of the formal metamathematics stand out.

Finally, rather than choosing a specific metalogic which is adequate (such as Nuprl or CoC), we talk about the requirements for such a theory.

2 Data Abstraction in Type Theory

In this section we provide a brief account of data abstraction within type theory. Our notion of data abstraction has similarities to two standard techniques used to define data within type theories; these are the techniques of defining data-types within minimal type theories such as CoC, and the definition of ADTs as Σ-types within predicative type theories.

Within minimal type theories, such as the Calculus of Constructions, standard data-types (e.g., pairing, lists, etc.) are encoded by inductive definitions. These data-types are specified by:

- naming a set (type) that members will inhabit;
- giving function constants (and their types) for constructing members of the set;
- providing rules for reasoning about type members and functions defined over them.

These declarations can be seen as capturing the rules for type formation, introduction, elimination, and computation. In theorem provers for CoC such as *LEGO*[9] these declarations are made within a *context*, and one reasons within the scope of these declarations. Theorems proved within contexts essentially have all of the context declarations (implicitly) quantified using Π-types.

When a type theory is enriched with Σ-types, a second approach is possible whereby the declarations of the ADT are packaged together with a (iterated) Σ-type; theorems are then proved within the context of these types. This has the conceptual advantage of unifying the

$$
\begin{array}{l}
LIST \mathrel{\hat{=}} A{:}Type \rightarrow \\
\quad List{:}Type \\
\quad \#\ Nil{:}List \\
\quad \#\ Cons:\ A \times List \rightarrow List \\
\quad \#\ Lind{:}\ P{:}(List \rightarrow Type) \rightarrow l{:}List \rightarrow P(Nil) \\
\qquad \rightarrow ((h{:}A \times t{:}List \times P(t)) \rightarrow P(Cons(h,t))) \rightarrow P(l) \\
\quad \#\ P{:}(List \rightarrow Type) \rightarrow l{:}List \rightarrow g{:}P(Nil) \rightarrow i{:}((h{:}A \times t{:}List \times P(t)) \rightarrow P(Cons(h,t))) \\
\qquad\qquad Lind(P)(Nil)(g)(i) = g \\
\qquad\qquad \wedge\ \forall h{:}A.\,\forall t{:}List.\, Lind(P)(Cons(h,t))(g)(i) = i(h,t,Lind(P)(t)(g)(i))
\end{array}
$$

$$
\begin{array}{l}
MSET \mathrel{\hat{=}} A{:}Type \rightarrow \\
\quad MSet{:}Type \\
\quad \#\ Empty{:}MSet \\
\quad \#\ Add:\ A \times MSet \rightarrow MSet \\
\quad \#\ \forall a,b{:}A.\,\forall s{:}MSet.\, Add(a,Add(b,s)) = Add(b,Add(a,s)) \\
\quad \#\ Mind{:}\ P{:}(MSet \rightarrow Type) \rightarrow l{:}MSet \rightarrow P(Empty) \\
\qquad \rightarrow \{i{:}((a{:}A \times s{:}MSet \times P(s)) \rightarrow P(Add(a,s))) \mid \\
\qquad\quad \forall a,b{:}A.\,\forall s{:}MSet.\,\forall p{:}P(s).\, i(a,Add(b,s),i(b,s,p)) = i(b,Add(a,s),i(a,s,p))\} \\
\qquad \rightarrow P(l) \\
\quad \#\ P{:}(MSet \rightarrow Type) \rightarrow l{:}MSet \rightarrow g{:}P(Empty) \\
\qquad \rightarrow i{:}\{i{:}((a{:}A \times s{:}MSet \times P(s)) \rightarrow P(Add(a,s))) \mid \\
\qquad\quad \forall a,b{:}A.\,\forall s{:}MSet.\,\forall p{:}P(s).\, i(a,Add(b,s),i(b,s,p)) = i(b,Add(a,s),i(a,s,p))\} \\
\qquad\qquad Mind(P)(Empty)(g)(i) = g \\
\qquad\qquad \wedge\ \forall a{:}A.\,\forall s{:}MSet.\, Mind(P)(Add(a,s))(g)(i) = i(a,s,Mind(P)(s)(g)(i))
\end{array}
$$

Figure 2: List and Multi-set ADT

data-type definition into a single type and the practical advantage of allowing parameterization of one ADT by another. It can also be seen as a natural generalization of signatures in ML, Extended ML [40], and other languages implementing data-abstraction using dependent types. This is the approach we shall take. One complication is that such modularization requires predicative quantification and hence a type hierarchy. For simplicity, we shall use the first universe (which we call *Type*) as the type of ADT parameters and set carriers.

Figure 2 contains two examples of ADTs: parameterized lists and finite multi-sets (sometimes called "bags"). Members of the LIST type are functions: when applied to a parameter type A, they return a tuple in which the first projection, the *carrier* of the ADT, is the type of lists over A whose members are built from the functions inhabiting the Nil and $Cons$ projections. The multi-set ADT is similar except for the defined carrier equality and the elimination and computation rules which are modified to respect this equality (more will be said on this below). Some notation should be explained. First $x : A\#B$ denotes a Σ-type built from the types A and B. Free occurrences of x become bound in B. Sometimes, to emphasize its logical significance, we abbreviate this to $\exists x : A.\, B$, or conjunction (when x is not free in B) $A \wedge B$, or even pairing $A \times B$. Similarly $x : A \rightarrow B$ represents the Π-type built from A and B and may be alternatively displayed as a $\forall x : A.\, B$ or $A \Rightarrow B$. To axiomatize equalities we make use of the equality (or I-type) of Martin-Löf type theory; equality is a ternary type-constructor ($t_1 = t_2 \; in \; T$) that has a member precisely when t_1 and t_2 are equal members of T. In our presentations we shall omit the third argument (the type T) since it can generally be determined from the context. We also make use of the set type $\{x : A | B\}$; in this type x is bound in B and its members are a in A when $B[a/x]$ is also inhabited. The proof rules for this type may be found in [12]. Finally, $\hat{=}$ is our notation for definitional equality.

The difference between these two ADTs is their specification of member equality. In an implementation of lists, members must be equal when they are built in identical ways from constructors. Indeed, when the induction combinator is known to be unique (which it is when the metalogic contains identity types), all implementing structures will be equivalent up to isomorphism. If we are working within a type theory such as Nuprl's which has a type constructor *list* with term constructors *nil* and "." (for cons) and an induction combinator *list_ind*, then we can trivially build a member of the list ADT.

$$\lambda A. \langle A \; list, \langle nil, \langle \lambda a. \lambda t. (a.t), \langle \lambda P. \lambda l. \lambda g. \lambda i. \, list\_ind(l; g; h, l, v.i(h)(l)(v)), \langle \lambda P. \lambda l. \lambda g. \lambda i. \langle axiom, \lambda a. \lambda t. \, axiom \rangle \rangle \rangle \rangle \rangle \rangle$$

Other implementations are possible; for example, lists as iterated pairs, fringes of trees, and so forth. For multi-sets, any implementation of the MSET ADT must equate not only those members built in the same way, but also those sets which only differ in the order in which their members were added. Specifying carrier types for ADTs that have axiomatized equalities requires a quotient type constructor such as the one provided by Nuprl, or extensions to the underlying type theory in the spirit of those suggested by Backhouse [3]. For example, using the quotient type we can implement multi-sets as lists where the quotienting equality is equivalence under permutation.

One complication of packaging ADT declarations as Σ-types is that we must be able to open them and access the components of their members. To this end, projection functions must be created and named, but these may be automatically generated in a straightforward way. In subsequent sections we shall always assume that these projection functions have been defined and are named, when possible, by the binding variable associated with that component of the Σ-type. For example, let L be member of $LIST$, then for T a type, $L(T)$ is a tuple with $List(L(T))$ naming the first projection, $Nil(L(T))$ naming the second, and so forth. To simplify presentation in the remainder of the paper we will often make radical abbreviations in presenting projections from ADTs, and terms built from the constructor functions. For example, when types like L and T can be determined from context we will write terms like $List(L(T))$ and $Nil(L(T))$ more simply as $List$ and Nil. Moreover we will use standard abbreviations in terms built from these constructors; for example we might abbreviate $Cons(L(T))(t_1, Cons(L(T))(t_2, Nil(L(T))))$ as $Cons(t_1, Cons(t_2, Nil))$, or even $[t_1, t_2]$.

The most significant distinction between our ADTs and traditional ADTs is that we axiomatize induction (elimination) rules and computation rules. The addition of these rules is not difficult within either impredicative or predicative type theories and may be automatically generated from the constructor declarations. In [37], Paulin-Mohring details how induction and computation principles can be automatically generated from declarations of constructor functions and their types and arities within the Calculus of Constructions; The INRIA implementation of CoC contains such a facility. In [3], Backhouse gives a similar account for predicative type theories (e.g., Martin-Löf's) and he also explains how these rules can incorporate quotiented equality relations like the kind needed for multi-sets. For brevity of presentation, we will not explicitly give the induction and computation rules associated with defined ADTs.

But these should be understood as being part of all ADTs defined.

As an example, within the context of the list ADT we may define the functions "head" and "tails" as follows.

$$\begin{array}{rcl} hd(l) & \hat{=} & Lind(\lambda x.\, List)(l)(Nil)(\lambda\langle h,v,t\rangle .\, h) \\ tl(l) & \hat{=} & Lind(\lambda x.\, List)(l)(Nil)(\lambda\langle h,v,t\rangle .\, t) \end{array}$$

From these definitions, using the *Lind* induction principle it is straightforward to prove theorems such as

$$\vdash \forall A{:}Type.\, \forall L{:}LIST.\, \forall l{:}List.\, l \neq Nil \Rightarrow Cons(hd(l), tl(l)) = l.$$

There are a couple subtleties that arise in the definition and application of functions over the ADT carriers. The first concerns the use of equality axioms in ADTs like the multi-set. Any defined functions must respect this equality and, as a result, many functions over lists, such as head and tails, have no analog for multi-sets since they do not respect the multi-set equality. This is sensible; since multi-sets are unordered, they have no "first" and "rest". Functions respecting this equality can still be defined. For example, we can define multi-set membership provided we are careful (e.g., through the use of the set type constructor) that the definition does not provide a way of computing the "position" of the element in the multi-set. In future sections, we shall make use of defined functions on both lists and multi-sets such as member deletion (which we denote by "$-$") and membership ("$\in$"). We shall also denote the empty list (multi-set) by "$\{\}$" and *cons* (*add*) by "$+$".

Another subtlety concerns equality reasoning. Note that the computation rules allow us to reason about the the conversion behavior of defined functions; we may compute with them, but only in an indirect sense. That is, the computation axioms do not augment the framework logic with reduction rules that can directly be computed with (in the sense of, say, the direct computation rules of Nuprl). Instead, one computes by using equality reasoning to establish that a defined function is equal to some normal form, e.g. $hd([1,2,3]) = 1$. This kind of computation via rewriting can be automated and made fairly efficient. In [25], Howe gives an example of how this kind of equality simplification can be automated by rewrite tactics similar to Paulson's conversions [38] and moreover how this can be accomplished (using derived inference rules) without direct construction of the corresponding equality proof. It would be interesting to study an alternative approach to computational equality in which direct computation rules are expressed in the ADT, perhaps in a type theory that contains a primitive for Howe's lazy congruence relation [26].

3 Specifying Provers With Abstract Types

Textbook presentations of logics typically begin with an account of logical syntax, i.e., the presentation of syntactic entities such as terms, formulas, and proofs. Typical is the text of Gallier [20], which presents the formulas of the propositional calculus as those in the inductive closure of a collection of propositional functions. As this is a freely generated set, he observes that we can define functions on formula by recursion and prove properties of formula by induction. Van Dalen's presentation [42] is similar, although he doesn't explicitly refer to the propositional connectives as functions.

What these and most other presentations have in common is that they define logical syntax as sets built from functions having certain properties. Equally significant, they leave implementation details unspecified. It should not matter whether formulas are represented by ASCII

$$\frac{}{\Gamma, A \vdash A}\,Axiom \qquad \frac{}{\Gamma, f \vdash G}\,FalseE$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,AndI \qquad \frac{\Gamma, A, B \vdash G}{\Gamma, A \wedge B \vdash G}\,AndE$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,OrIL \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,OrIR \qquad \frac{\Gamma, A \vdash G \quad \Gamma, B \vdash G}{\Gamma, A \vee B \vdash G}\,OrE$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\,ImpI \qquad \frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A \Rightarrow B \vdash G}\,ImpE$$

Figure 3: LJ Inference Rules

strings or trees built from cons cells; all that matters is their extensional properties. This kind of abstraction and desire for representation independence between the specification of a logic and its implementation suggests that they stand in the same relation as that between an abstract data-type or signature (in the parlance of ML modules [35]) and its implementing structure. Note, in particular, that the induction and recursion requirements are perfectly captured by our induction and computation rules.

In this section we illustrate this approach to specification by formalizing a theorem prover for a simple logic: a sequent calculus presentation of an intuitionistic propositional calculus. This system, LJ, is taken directly from Dyckhoff [16] and is due to Gentzen [21]. Figure 3 contains a bottom-up presentation of the rules of this logic. We have chosen LJ for two reasons. First, as it is simple it admits an exceptionally clean exposition. So the reader won't feel cheated, in Appendix A we indicate how the term and proof types can be generalized in a straightforward way to the more complex predicate calculus. But we have a second reason, and that is a simple variant of this logic has certain useful structural properties that we will use to illustrate metatheoretic extensibility via proof-transformations in Section 5.

3.1 Well Formed Formula

Well formed formula (Wffs) of LJ are those terms freely generated from a collection of variables and propositional constructors. We shall assume that the set of propositional constructors is fixed in advance, although this restriction is not strictly necessary. The Wff and proof data-types could be parameterized over sets of Wff and proof constructors and this can lead to a kind of theory-polymorphic metatheory. We do not explore this possibility though.

To specify variables, we will take as given in the framework logic a type *Atom* which is a collection of names that may be associated with variables.[1] The decision not to represent variables using variables in the framework logic is deliberate and will be discussed in detail in Section 6 where we compare this to the alternative, which is used in frameworks like ELF. The

[1]This may be a base type of strings in the framework logic (e.g., the type of strings *Atom* in Nuprl's type theory), or alternatively defined as a type of elements whose equality relation is decidable. We take such an approach to defining function and predicate names in the appendix.

$$
\begin{aligned}
&\mathit{WFF} \mathrel{\hat{=}} \\
&\qquad \mathit{Wff}\!:\!\mathit{Type} \\
&\quad \#\ \mathit{Var}\!:\!\mathit{Atom} \to \mathit{Wff} \\
&\quad \#\ \mathit{False}\!:\!\mathit{Wff} \\
&\quad \#\ \mathit{And}:\ \mathit{Wff} \times \mathit{Wff} \to \mathit{Wff} \\
&\quad \#\ \mathit{Or}:\ \mathit{Wff} \times \mathit{Wff} \to \mathit{Wff} \\
&\quad \#\ \mathit{Imp}:\ \mathit{Wff} \times \mathit{Wff} \to \mathit{Wff}
\end{aligned}
$$

Figure 4: Propositional Formula ADT

formalization of this as an ADT is given in Figure 4. Recall that this ADT, like all our ADTs, also comes equipped with the appropriate induction and computation principles.

3.2 LJ Inference Rules and Proofs

We have considerable flexibility in how we present proofs as an ADT. In this section we present a simple proof type for LJ and indicate alternatives that differ in their assumptions about the structure of proofs. Alternatives formalizations are significant as different types capture different models (implementations) of theorem proving and support different kinds of metatheoretic reasoning.

We begin with a simple encoding that captures a minimal amount of proof structure; essentially, the relationship between sequents that comprise proofs. It will have some similarities to ELF-style encodings since both explain the meaning of logical constants in the encoded logic through the declarations of constants in the framework logic. However unlike ELF encodings, where each proposition p has its own type of its proof-objects, $Tr(p)$, we define a single type of proofs which we can equip with induction and computation principles.

To represent sequent calculus inference rules we must first define a data-type of sequents. In the system we formalize, a sequent is a pair where first component represents hypotheses and the second the conclusion. Furthermore, hypotheses are unordered and duplication is allowed; this eliminates the need for structural rules. Hence we model hypotheses using multi-sets of Wffs. The second component is simply a Wff.

$$
\begin{aligned}
&\mathit{SEQUENT} \mathrel{\hat{=}} W : \mathit{WFF} \to M : \mathit{MSET} \to \\
&\quad \mathit{Sequent}\!:\!\mathit{Type}\ \#\ \ \mathit{MkSeq}\!:\!\mathit{MSet}(M(\mathit{Wff}(W))) \times \mathit{Wff}(W) \to \mathit{Sequent}
\end{aligned}
$$

Note that due to parameterization, any implementation of sequents depends on how both the Wff and multi-set ADT parameters are instantiated, and of course, the function inhabiting the sequent signature. Also note that by analysis of the component ADTs we may determine what principles may be employed to reason about and compute with sequents. For example, we may split sequents into their components, perform Wff induction on the conclusion, perform multi-set induction on the hypothesis set, and so forth. For s a member of the carrier of this type, we will denote its first and second projections by $\mathit{Hyps}(s)$ and $\mathit{Concl}(s)$ respectively.

The LJ proof rules (Figure 3) specify a relationship between tuples of sequents (the premises of the rule) and a sequent (the conclusion). Proof rules have limited applicability and we express constraints and side conditions using dependent Π-types and set-types in the framework logic. The use of set types here is not strictly necessary (one could achieve similar specifications

$$
\begin{array}{l}
LJ\_PROOF \mathrel{\hat{=}} W\!:\!WFF \rightarrow M\!:\!MSET \rightarrow S\!:\!SEQUENT \rightarrow \\
\quad\quad LJ\!:\!Type \\
\quad \#\ Seq:\ LJ \rightarrow Sequent \\
\quad \#\ Axiom:\ s_1\!:\!\{s\!:\!Sequent|Concl(s) \in Hyps(s)\} \rightarrow \{p\!:\!LJ|Seq(p) = s\} \\
\quad \#\ FalseE:\ s_1\!:\!\{s\!:\!Sequent|False \in Hyps(s)\} \rightarrow \{p\!:\!LJ|Seq(p) = s\} \\
\quad \#\ AndI:\ p_1\!:\!LJ \\
\quad\quad\quad \rightarrow p_2\!:\!\{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1))\} \\
\quad\quad\quad \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = And(Concl(Seq(p_1)), Concl(Seq(p_2)))\} \\
\quad \#\ AndE:\ w_1\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow w_2\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow p_1\!:\!\{p\!:\!LJ|w_1 \in Hyps(Seq(p)) \wedge w_2 \in Hyps(Seq(p))\} \\
\quad\quad\quad \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) + And(w_1, w_2) - w_1 - w_2 \wedge Concl(Seq(p)) = Concl(Seq(p_1))\} \\
\quad \#\ OrIL:\ w_1\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow p_1\!:\!LJ \\
\quad\quad\quad \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = OR(Concl(Seq(p_1)), w)\} \\
\quad \#\ OrR:\ w_1\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow p_1\!:\!LJ \\
\quad\quad\quad \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = OR(w_1, Concl(Seq(p_1)))\} \\
\quad \#\ OrE:\ w_1\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow w_2\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow p_1\!:\!\{p\!:\!LJ|w_1 \in Hyps(Seq(p))\} \\
\quad\quad\quad \rightarrow p_2\!:\!\{p\!:\!LJ|w_2 \in Hyps(Seq(p)) \\
\quad\quad\quad\quad\quad\quad \wedge\ Hyps(Seq(p)) - w_1 = Hyps(Seq(p_2)) - w_2 \wedge Concl(Seq(p)) = Concl(Seq(p_2))\} \\
\quad\quad\quad \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) - w_2 + Or(w_1, w_2) \wedge Concl(Seq(p)) = Concl(Seq(p_1))\} \\
\quad \#\ ImpI:\ w_1\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow p_1\!:\!\{p\!:\!LJ|w_1 \in Hyps(Seq(p))\} \\
\quad\quad\quad \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) - w_1 \wedge Concl(Seq(p)) = Imp(w_1, Concl(Seq(p_1)))\} \\
\quad \#\ ImpE:\ w_1\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow w_2\!:\!W\!f\!f \\
\quad\quad\quad \rightarrow p_1\!:\!\{p\!:\!LJ|Imp(w_1, w_2) \in Hyps(Seq(p)) \wedge w_1 = Concl(Seq(p))\} \\
\quad\quad\quad \rightarrow p_2\!:\!\{p\!:\!LJ|Hyps(Seq(p_1)) - Imp(w_1, w_2) + w_2 = Hyps(Seq(p))\} \\
\quad\quad\quad \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1))) \wedge Concl(Seq(p)) = Concl(Seq(p_2))\}
\end{array}
$$

Figure 5: LJ ADT

using additional axioms to capture these constraints) however they do provide an economy of expression as well as a way of tagging computationally uninteresting parts of the specification. For example, if LJ is the type of proofs, and for p in LJ, $Seq(p)$ represents the sequent proved, then we can formalize $AndI$ in the following way.

$$
\begin{array}{rl}
AndI:\ p_1\!:\!LJ \rightarrow & p_2\!:\!\{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1))\} \\
& \rightarrow \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) \\
& \quad \wedge\ Concl(seq(p)) = And(Concl(Seq(p_1)), Concl(Seq(p_2)))\}
\end{array}
$$

This rule takes two members of the proof-type, p_1 and p_2 where the sequent proved in p_2 has the same hypotheses as the sequent proved in p_1, and returns a member of the proof-type with the same hypotheses as p_1 and a conclusion that is the conjunction of the conclusions of p_1 and p_2. Other rules are presented in Figure 3.2.

Within the context of this proof type we may formulate the type of provable Wffs as follows.

$$Thm(w) \mathrel{\hat{=}} \{p\!:\!LJ|Seq(p) = MkSeq(Nil, w)\}$$

Similar in spirit to the ELF, the inhabitants of the carrier LJ encode proofs. For example, if A

is the propositional variable $Var(W)(\text{"A"})$, and similarly for B, then

$$ImpI(A, ImpI(B, axiom(MkSeq([A, B], A)))$$

is a member of $Thm(Imp(A, ImpI(B, A)))$.

The LJ proof type specifies a bottom-up style of proof-construction where one works backwards from theorems (under assumptions) to theorems (eventually without assumptions). Of course no orientation is specified by the informal presentation of these proof-rules in Figure 3, and we have the freedom to formalize alternative proof presentations of these rules. Different proof types will capture different models of theorem proving and different metatheories.

To illustrate, let us briefly sketch an alternative formulation, an ADT for a (slightly novel) kind of top-down proofs. The carrier of the ADT will have as members proofs constructed by refinement of goals. Refinement proceeds by the user naming a goal and applying a proof rule; the result is 0 or more subgoals. Goals will not be organized as a tree or stack, but rather as an unordered set (so identical unproven subgoals will be collapsed). Each goal in the set will be a *marked* sequent, which is a sequent paired with a status indicator of whether it has been refined (*Ref*) or is still unrefined (*UnRef*). A proof begins with an injection of a Wff into the proof type (with a function we call *Goal*) and ends when all sequents have been refined. Figure 3.2 contains a partial definition of this proof type. Here $MarkedSequents$ is the carrier of an ADT representing a set of sequents with status. This type has a constructor $Mark$ that given a member of the Sequent type (built with $MkSeq$) and a status indicator returns a marked sequent. The marked sequent ADT also has defined operators for member addition and deletion. The function $IsConjunct$ is defined by recursion on Wffs and is true when its argument is a conjunction. Similarly $LeftConjunct$ and $RightConjunct$ return the left and right conjuncts of a Wff that represents a conjunction.

Unlike the LJ ADT not all elements of this proof type represent theorems. The theorem type for this ADT could simply be formalized as

$$Thm(w) \mathrel{\hat{=}} \{p{:}Top | \forall m{:}MarkedSequents.\, m \in SubGoals(p) \Rightarrow Refined(m)\}$$

where the proposition $Refined(m)$ is true when the sequent m is marked as refined.

Formalizing different models of theorem proving is interesting in its own right; it can bring into focus *how* the logic will be used in theorem proving. Assumptions can be made explicit such as whether proofs are constructed top-down, bottom-up, or in no particular order, whether they are constructed sequentially or in parallel, whether proof are represented by trees, DAGS, stacks, and so forth. Metatheoretically, as we alter the proof type, we change what concepts we can express and what properties of encoded proofs we can prove in our framework logic. For example, within the context of the top-down proof type we can reason about incomplete proofs and routines (e.g., tactics) that extend and possibly complete these proofs. This is not possible with our LJ proof type. Our belief is that, in the broad sense of the word, the "metatheory" of a logic can go far beyond simply a theory of derivable rules and the more of a logic that we formalize the better we can reason about it.

4 Derived Rules

In the previous section we demonstrated that given our encoding of proof terms we can formulate a type of theorems in a straightforward way. All members of the LJ proof type are theorems, and members of TOP are theorems when all sequents are marked as refined. In both cases,

$$
\begin{array}{l}
\quad Top : Type \\
\# \ SubGoals : \ Top \to MarkedSequents \\
\# \ Goal : \ w : Wff \\
\quad \to \{p : Top | SubGoals(p) = \{\} + Mark(MkSeq(Nil, w), UnRef)\} \\
\# \ Axiom : \ s_1 : \{s : Sequent | Concl(s) \in Hyps(s)\} \\
\qquad \to p_1 : \{p : Top | Mark(s, UnRef) \in SubGoals(p)\} \\
\qquad \to \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s, UnRef) + Mark(s, Ref)\} \\
\# \ FalseE : \ s_1 : \{s : Sequent | False \in Hyps(s)\} \\
\qquad \to p_1 : \{p : Top | Mark(s, UnRef) \in SubGoals(p)\} \\
\qquad \to \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s, UnRef) + Mark(s, Ref)\} \\
\# \ AndI : \ s_1 : \{s : Sequent | IsConjunct(Concl(s))\} \\
\qquad \to p_1 : \{p : Top | Mark(s, UnRef) \in SubGoals(p)\} \\
\qquad \to \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s, UnRef) + Mark(s, Ref) \\
\qquad\qquad\quad + Mark(MkSeq(Hyps(S), LeftConjunct(Concl(s_1))), UnRef) \\
\qquad\qquad\quad + Mark(MkSeq(Hyps(S), RightConjunct(Concl(s_1)), UnRef)))\} \\
\# \ AndE : \ w_1 : Wff \\
\qquad \to w_2 : Wff \\
\qquad \to s_1 : \{s : Sequent | And(w_1)(w_2) \in Hyps(s)\} \\
\qquad \to p_1 : \{p : Top | Mark(s, UnRef) \in SubGoals(p)\} \\
\qquad \to \{p : Top | Subgoals(p) = Subgoals(p_1) - Mark(s, UnRef) + Mark(s, Ref) \\
\qquad\qquad\quad + \ Mark(MkSeq(Hyps(S) - And(w_1, w_2) + w_1 + w_2, Concl(s_1)), UnRef)\} \\
\ldots Other \ Proof \ Rules \ Similar
\end{array}
$$

Figure 6: Top Down “Unstructured” LJ ADT

we can construct proofs for the encoded logic by demonstrating that these theorem types are inhabited in the framework logic. This is similar to the conventional uses of type theory as a logical framework. In this section we shall indicate how representing logics with ADTs provides a basis for metatheoretic reasoning that extends beyond simply capturing provability.

The most common method of augmenting an implementation of a logic with metatheoretic capabilities is to extend it with a metalanguage for writing *tactics*, which are programs that construct and manipulate proofs. Many theorem proving systems, for example LCF, HOL, and Nuprl, take this approach. Each is equipped with a programming language (in all three cases ML), and an interface between this language and the proof engine. This interface typically consists of types for object-logic syntax (e.g., terms, sequents, proofs) and functions that manipulate and construct members of these types. This approach to metareasoning can be directly used within our framework when the implementation of the framework logic supports a metalanguage. This capability is not unique to our approach and is shared by other approaches to logical encoding.

But there is an alternative that appears to be possible when two conditions are met. Specifically,

1. the framework logic is a programming logic, and
2. logics have been encoded in a way that renders it possible to reason about the encodings and define functions over them.

When the first condition is satisfied, which it is when the framework logic is a constructive type theory, then the logic itself is expressive enough to serve as both its own metalanguage and metalogic. When the second condition is also satisfied, which it is when induction and computation axioms are provided for the encoded data-types, we can use this metalogic to reason about the logical encodings and functions defined over these encodings. Functions may be defined explicitly (using the appropriate axiomatized recursion combinators), or alternatively built implicitly by constructive inductive proofs (e.g., the proofs-as-programs paradigm). These functions can implement general kinds of derived inference rules including term normalizers, decision procedures, and the like.

The approach to constructive metatheory we are proposing is very similar in spirit to the kind of metatheory developed in [24, 13] in which term rewriters, matchers, and normalizers, were implemented and formally reasoned about within the Nuprl type theory. The primary difference is that in the Cornell work, the metatheory was about a specific encoding of terms (e.g., Nuprl terms in [24]) and to be used in Nuprl theorem proving a connection had to be made between the encoded terms and Nuprl's terms. Under our approach, data-types such as terms and proofs are specified within the context and metatheorems are never used at the "object level" (e.g., we never work with implementations of the encoded logic) so there are no difficulties involving reflection or moving between syntactic entities and their encodings. Derived rules are simply theorems about properties of the relevant proof types and functions defined over these types and are used to construct and manipulate members of these types within the framework logic. (More will be said on these points in Section 6.)

In this section we provide a simple example: a derived inference rule about the provability of a subclass of propositional Wffs. We take this example directly from Weyhrauch [43] where he states:

> If you have a propositional Wff whose only sentential connective is the equivalence sign, then the Wff is a theorem if each sentential symbol occurs an even number of times.

Weyhrauch stated his theorem for a logic in which the Wff type includes the "iff" connective (which we denote by the prefix binary relation *Iff*) and the proof system is classical. We can easily extend our Wff type (with this connective) and LJ proof type (with the rule of excluded middle) to encode this desired logic. Let $\mathcal{C}$ denote the context containing declarations of these types as well as definitions of the supporting ADTs. Furthermore, let *parity* be a function of type $\mathit{Wff} \rightarrow \{0..1\}$ that returns 1 exactly when its argument is built only from the *Iff* connective and variables, and furthermore, each variable occurs exactly an even number of times. This can be defined using the Wff recursion combinator in a straightforward way.

Now, within this context, we may formalize Weyhrauch's theorem as

$$\mathcal{C} \vdash \forall w{:}\,\mathit{Wff}.\,\mathit{parity}(w) = 1 \Rightarrow \mathit{Thm}(w).$$

Let us sketch a constructive proof of this by induction on the complexity of the expression w. Given a Wff w such that $\mathit{parity}(w) = 1$, then w is a term built from *Iff* that must contain at least two occurrences of the same variable. Call this variable a; there are two cases. The first is $w = \mathit{Iff}(a, a)$, but then we are done as $\mathcal{C} \vdash \mathit{Thm}(\mathit{Iff}(a, a))$ is provable. In the second case, w must contain more than one *Iff* connective. Then, because *Iff* is associative and commutative, we can rearrange and reassociate the variables in w to "shuffle" together two occurrences of a. More formally, there must be Wff w' such that $\mathit{parity}(w')$ and

$$\mathcal{C} \vdash \mathit{Thm}(\mathit{Iff}(\mathit{Iff}(a, a), w') \Leftrightarrow \mathit{Thm}(w).$$

Furthermore, we can "cancel" the $\mathit{Iff}(a, a)$, that is

$$\mathcal{C} \vdash \mathit{Thm}(w') \Leftrightarrow \mathit{Thm}(\mathit{Iff}(\mathit{Iff}(a, a), w')).$$

Since $parity(w') = 1$ and w' is simpler (e.g., fewer variables) than w, we have $Thm(w')$ by the induction hypothesis. Hence, we can conclude $Thm(w)$.

The above proof sketch should be formalizable within our framework logic in a straightforward way. Formalizing the theorem statement requires that we can define the parity function by recursion over the structure of Wffs. This is not difficult and can for example be implemented by a function which makes several passes over its Wff argument: first to check the connective structure and second to gather up the variables at the leaves to verify that each occurs an even number of times. Formalizing the proof requires reasoning about Wffs (and the parity) function) using the Wff induction principle. In particular, we must be able to derive alternative induction schemas such as induction over well-founded measures on Wffs. In the above proof, this measure would be the number of variables in the formula. Complete induction over this well-order (although not all well-orders) is interderivable with our axiomatized structural induction principle for Wffs.

A proof of the parity theorem will (within a constructive framework logic) construct a function that itself constructs proofs. Specifically, let *par* be a function inhabiting the type of the parity theorem and let w be a member of *Wff* such that $parity(w) = 1$. The existence of *par* formally asserts that w is provable and we may use this assertion in other proofs. Moreover, should we actually require the proof of w, we may generate it by evaluating *par*.

5 Proof Theory

The implementation of the object theory lets one investigate whether a specific proposition is provable. The metatheory lets us investigate whether there is a general method of settling the provability of a class of propositions, and whether there is even a *good* method for doing this.

For the IPC, Intuitionistic Propositional Calculus, there are well-known algorithms to decide provability. The first is due to Gentzen, based on his Hauptsatz, but it is not a good method. Basically he suggests searching the space of all possible proofs up to a certain depth. We can consider this space to be an infinite tree whose nodes are sets of sequents and whose edges are labeled by the rule name used to replace a sequent, call it the goal, by a finite number of subgoal sequents. We are thinking then of generating proofs *top-down*, and it is natural to use a top-down proof type. This tree of all possible proofs will contain any proof that exists, and if we knew that any proof must occur within a certain depth, then we could just search to that depth, which is a finite process. But indeed we do know from the Hauptsatz that we only need to search to a finite depth. Namely we know that if there is a proof, then there is one which uses only subformulas of the goal. Moreover, if in the search for a proof in the tree we encounter a node whose set of sequents appears earlier on the same path, then we can terminate the search along that path. Since only subformulas can appear in the sequents, there are only finitely many different nodes that can occur in this tree, so along any path, either we find a proof or repeat a node thus ending the search. Moreover we can precompute the depth at which this repetition must occur, so the argument is constructive.

For the classical propositional calculus, PC, there is a much simpler proof search strategy. First every rule can be seen to decompose a goal sequent into simpler sequents in that the formulas occurring in the subgoal sequents are all of smaller complexity than those in the goal. So the termination argument can be given in terms of a multiset ordering on the sets of sequents occurring at the nodes in the tree of all possible proofs. In addition, there is a fixed order for applying the rules which is guaranteed to work if any order does, so we need search only one path

$$\frac{\Gamma, C \Rightarrow (D \Rightarrow B) \vdash G}{\Gamma, (C \wedge D) \Rightarrow B \vdash G}\, ImpE_1 \qquad \frac{\Gamma, C \Rightarrow B, D \Rightarrow B \vdash G}{\Gamma, (C \vee D) \Rightarrow B \vdash G}\, ImpE_2$$

$$\frac{\Gamma, D \Rightarrow B \vdash C \Rightarrow D \quad \Gamma, B \vdash G}{\Gamma, (C \Rightarrow D) \Rightarrow B \vdash G}\, ImpE_3 \qquad \frac{\Gamma \vdash G}{\Gamma, False \Rightarrow B \vdash G}\, ImpE_4$$

$$\frac{\Gamma, B, A \vdash G}{\Gamma, A \Rightarrow B, A \vdash G}\, ImpE_5 \qquad \textit{(for A atomic)}$$

Figure 7: LJT Rules Replacing *ImpE*

in this tree. With these observations one is led to the tableau search procedure for example.

It is natural to ask whether there is a similar proof search procedure for the IPC. The complication is that Gentzen's original system, LJ, uses the implication elimination rule

$$\frac{H, A \Rightarrow B \vdash A \quad H, B \vdash G}{H, A \Rightarrow B \vdash G}$$

and the subgoal sequent $H, A \Rightarrow B \vdash A$ might in fact be more complex than the goal sequent if A is more complex than G. So one cannot use a multiset ordering. Various methods have been discovered to circumvent this difficulty. One can compute the number of times the formula $A \Rightarrow B$ needs to be used for example. Roy Dyckhoff has discovered a method [16], and in his paper he surveys other ones and recommends the source [15]. His idea is to replace the implication elimination rule with the five others given in Figure 7; he calls the resulting in the system he calls LJT In LJT all rules generate simpler subgoal sequents so that termination of the proof search can be guaranteed via the multiset ordering. He proves:

A formula of IPC is provable in LJ iff it is provable in LJT.

Here is a metamathematical result which is very useful. It is the basis for a good decision procedure for the IPC. Indeed Dyckhoff has implemented such an algorithm for the system MacLogic (see [17]). Suppose then that in an implemented object logic based on LJ one is looking for a decision procedure for IPC formulas. If we restrict ourselves to derived rules based on LJ, we will not find Dyckhoff's method. But if we can prove metatheorems like his in the metalogic, then we can use them. So for instance, from a constructive proof of his theorem, we can build a decision procedure which takes a formula G, applies Dyckhoff's procedure to find an LJT proof or report that there is no proof. If there is no LJT proof, then we know there is no LJ proof either, so we report failure. If there is an LJT proof, then by the constructive nature of the proof of the metatheorem, we can transform it to an LJ proof as required.

Indeed Dyckhoff's result can be proved constructively. We outline such a proof below and suggest in a bit more detail how a formalization of it in type theory is used. First define the depth of a formula as follows:

$$depth(A) = depth(False) = 1 \text{ for } A \text{ atomic},$$

$$depth(A \vee B) = depth(A \Rightarrow B) = depth(A) + depth(B) + 1,$$

$$depth(A \wedge B) = depth(A) + depth(B) + 2.$$

This introduces an ordering on formulas. We extend it to an ordering on multisets of formulas in the natural way, namely $B_1 \ll B_2$ iff B_1 is obtained from B_2 by replacing formulas in B_2 by a finite set of those of smaller depth.

We prove two separate theorems:

Theorem 1 *If a formula G of IPC is provable in LJ, then it is provable in LJT.*

Theorem 2 *If a formula G of IPC is provable in LJT, then it is provable in LJ.*

We only need Theorem 2 to be proved constructively, but in fact constructive proofs of each theorem are just as clear and compact as classical proofs and provide more information, so we prove them this way. For Theorem 1 we follow Dyckhoff and prove a lemma about how to postpone uses of implication elimination. First a definition.

Definition 1 *Call a proof* awkward *iff the final step is an elimination on $A \Rightarrow B$ where A is atomic and does not occur as an hypothesis (i.e. on the left side of the sequent).*

Lemma 1 *If $\Delta \vdash G$ is provable in LJ, then it has a nonawkward proof.*

Proof: Let P be a proof of $\Delta \vdash G$, proceed by induction on the structure of P, i.e., by the induction rule provided by the proof data type. If it is not awkward we are done, so suppose it is, and the last step is

$$\frac{\Gamma, A \Rightarrow B \vdash A \quad \Gamma, B \vdash G}{\Gamma, A \Rightarrow B \vdash G}$$

where A is atomic and does not occur in Γ. Let P_A be the subproof of A from Γ and $A \Rightarrow B$. This must end with an elimination step since A is atomic and does not occur in Γ. We consider all (four) ways that this can happen, which are: False-elim, $\wedge$-elim, $\vee$-elim and $\Rightarrow$-elim. In each case we show how to build a nonawkward proof.

1. If the rule is False-elim, then $False$ occurs in Γ. So we can use False-elim to prove $\Delta \vdash G$ in the first place. Call this proof P', it is not awkward.

2. If the rule is $\wedge$-elim, say on $C \wedge D$, then the proof looks like

$$\frac{\dfrac{\Sigma, C, D, A \Rightarrow B \vdash A}{\Sigma, C \wedge D, A \Rightarrow B \vdash A} \quad \Sigma, C \wedge D, B \vdash G}{\Sigma, C \wedge D, A \Rightarrow B \vdash G.}$$

Change this to the nonawkward proof P'

$$\cfrac{\cfrac{\Sigma, C, D, A \Rightarrow B \vdash A \quad \Sigma, C, D, B \vdash G}{\Sigma, C, D, A \Rightarrow B \vdash G}}{\Sigma, C \wedge D, A \Rightarrow B \vdash G.}$$

We know that there is a proof of $\Sigma, C, D, B \vdash G$ because we can cut in the formula $C \wedge D$ and use the given proof of $\Sigma, C \wedge D, B \vdash G$ known by our assumption on the shape of P. Then we obtain the proof P' we need by cut elimination on this subproof.

3. If the rule is $\vee$-elim, say on $C \vee D$, then the proof looks like

$$\cfrac{\cfrac{\Sigma, C, A \Rightarrow B \vdash A \quad \Sigma, D, A \Rightarrow B \vdash A}{\Sigma, C \vee D, A \Rightarrow B \vdash A} \qquad \Sigma, C \vee D, D \vdash G}{\Sigma, C \vee D, A \Rightarrow B \vdash G}$$

This can be converted to

$$\cfrac{\cfrac{\Sigma, C, A \Rightarrow B \vdash A \quad \Sigma, C, B \vdash G}{\Sigma, C, A \Rightarrow B \vdash G} \qquad \cfrac{\Sigma, D, A \Rightarrow B \vdash A \quad \Sigma, D, B \vdash G}{\Sigma, D, A \Rightarrow B \vdash G}}{\Sigma, C \vee D, A \Rightarrow B \vdash G}$$

The sequent

$$\Sigma, C, B \vdash G$$

can be proved by just cutting in $C \vee D$ which requires proofs of $\Sigma, C, B, C \vee D \vdash G$ and $\Sigma, C, B \vdash C \vee D$. The first one can be proved by weakening to remove the hypothesis C and then using the proof of $\Sigma, D, B \vdash G$. Call this proof $\hat{P}$. A similar modification is used to produce say $\bar{P}$ which proves $\Sigma, D, B \vdash G$.

4. If the rule is $\Rightarrow$-elimination, say on $C \Rightarrow D$, then the proof looks like

$$\cfrac{\cfrac{\begin{matrix} P_A \\ \Sigma, D, A \Rightarrow B \vdash A \end{matrix} \quad \begin{matrix} P_C \\ \Sigma, C \Rightarrow D, A \Rightarrow B \vdash C \end{matrix}}{\Sigma, C \Rightarrow D, A \Rightarrow B \vdash A} \qquad \begin{matrix} P_B \\ \Sigma, C \Rightarrow D, B \vdash G \end{matrix}}{\Sigma, C \Rightarrow D, A \Rightarrow B \vdash G}$$

Since we have already transformed the subproof of $\Gamma, A \Rightarrow B \vdash A$ to be nonawkward, we know that in the proof of $\Sigma, C \Rightarrow D, A \Rightarrow B \vdash A$, C is either nonatomic or occurs in Σ. We now convert the given proof P so that the last step is by elimination on $C \Rightarrow D$. This step is not awkward. This produces the left subgoal of showing $\Sigma, C \Rightarrow D, A \Rightarrow B \vdash A$ and the right subgoal of showing $\Sigma, D, A \Rightarrow B \vdash G$. The left subgoal can be proved from one of the subproofs of P and the right one is proved by elimination on $A \Rightarrow B$. One of the subgoals here is proved using a subproof of P and the other one, $\Sigma, D, B \vdash G$ is proved by cutting in $C \Rightarrow D$ and applying cut elimination to the resulting proof. □

Now Theorem 1 is proved by induction on the multi-set ordering using the lemma, and Theorem 2 is proved by induction on the proof structure using cut followed by cut elimination. The straight forward proofs here are constructive.

Let $LJTProof(G)$ denote those LJT proofs of G, likewise for $LJProof(G)$. Then Dyckhoff's decision procedure can be derived from a proof of the following theorem. First let S be a list of IPC formulas, say H_1,...,H_n, G. We regard it as a sequent with goal G and the other formulas as hypotheses and write $LJTProof(S)$ for the set of its LJT proofs.

Theorem 3 *Given any set $\{S_1, \ldots, S_n\}$ where S_i is a sequent over IPC formulas, we know for each S_i that $\exists p : LJTProof(S_i) \vee \neg\exists p : LJTProof(S_i)$.*

Corollary 1 *For all formulas G of IPC either there is an LJT proof of it or not.*

The corollary is proved by taking $\{G\}$ as the set of formulas in the theorem. The theorem is proved by the multi-set induction defined above. For each S_i we consider all possible rules r that can be applied, noting that each one decomposes S_i into $\{U_1, \ldots, U_k\}$ for which the induction hypothesis holds. If the U_j are all provable for some rule r, then S_i is. If not, then there is no rule for proving S_i, so there is no proof of it.

The constructive proof of Theorem 3 and its corollary provides a function from formulas into a disjoint union. If the result is in the left disjunct, then we can pull out a proof of the formula; if it is in the right disjunct, then we know there is no proof (and if we wanted to we could pull out the evidence for this conclusion). So Theorem 3 gives a decision procedure for provability and a proof generating procedure for true formuals. The constructive proofs of Theorem 2 and Theorem 1 provides functions $J : LJTProofs \Rightarrow LJProofs$ and $T : LJProofs \Rightarrow LJTProofs$. With these we can prove another corollary.

Corollary 2 *For all formulas G of IPC either there is an LJ proof of it or not.*

6 Comparison To Other Work

We conclude with a brief comparison to closely related work. First, we compare our framework logic with other frameworks, both in terms of the logic itself and how we encode logics within it. After, we discuss how our approach relates to other research in metatheory.

6.1 Framework Logics and Logical Encodings

Framework logics such as the ELF and Isabelle were designed with concerns different from metatheoretic reasoning and they are too weak to support it. Furthermore, as their concerns are different, logics are encoded within these frameworks in a way that does not support metatheoretic analysis. Let's address these points individually.

The logical basis of the framework we propose is a constructive type theory and in this respect it is most similar to the LF logic of the ELF system. However, LF was designed to be as weak as possible (e.g., one cannot abstract over members of *Type*) to make faithfulness and adequacy proofs easier. But this minimality only supports a very weak kind of metatheory, e.g., one can demonstrate that specific formulas have proofs. Both Taylor and Pollack, who have proved theorems about LF encodings (as reported in [41]), have had to formalize their proofs within the stronger Calculus of Constructions. Other framework logics, such as Isabelle and λ-Prolog, are based on higher-order intuitionistic logic and lack the propositions-as-types correspondence. Hence, demonstrations of provability do not construct proofs, and any kind of metatheoretic reasoning within the framework (none has been tried to our knowledge) would not construct functions over proofs. Furthermore, these logics do not provide a unified view of the relationship between the specification of a logic and its implementation.

Equally significant as differences in logics are differences in the way that they are used. Within the frameworks mentioned above one seeks to encode logics in a manner that *internalizes* parts of the structure of terms and proofs within the framework logic. By this we mean that syntactic notions and operations are subsumed by operations implemented in the framework logic. The alternative to this identification is to *externalize* these entities and operations by explicitly encoding syntax and functions over syntax.

The standard example of internalization is representing variable binding by λ-abstraction in the framework logic so that substitution can be implemented by β-reduction; supporting this internalization is one of principles behind the design of these framework logics. It is true that this has the advantage of liberating the logic encoder from concerns about substitution, bound variable renaming, and the like. But from a metalogical perspective, this choice seems inappropriate since it obscures the separation between the encoded logic and the framework logic; the terms in the framework logic which represent the terms and proofs of the encoded logic combine both meta and object level syntax. One can no longer view a logic's presentation as a specification of an implementation. More importantly, the logic's presentation cannot easily be the subject of metatheoretic analysis because analyzing variables and operations on them requires reasoning about the framework logic itself. It is no coincidence that most experiments in metatheoretic reasoning have picked algebraic problems (e.g., Taylor's associative term-equality example [41]) that do not require such analysis.

Internalization manifests itself in other ways as well. Consider for example, the difference between how we formulate inference rules in the LJ proof type (Section 3.2), and how these rules would be formulated in the ELF-style. In ELF, *AndI* would be formulated natural deduction style along the lines of

$$AndI : \forall A, B : \mathit{Wff}.\ \mathit{Tr}(A) \rightarrow \mathit{Tr}(B) \rightarrow \mathit{Tr}(And(A, B)).$$

This rule is simpler than ours, but that is because it effectively internalizes much of the structure of proofs; rather than formalizing and reasoning about an explicit representation of proofs, which requires explicitly formalizing sequents (in the case of the sequent calculus) or assumption dependencies and discharging (in the case of natural deduction) and the like, these concepts are instead captured by the meaning of implication in the framework logic. Again there are tradeoffs. This style of natural deduction presentation has advantages in that it allows rapid specification and prototyping of theorem provers. But again, it is unsuitable for our needs. First, it is not always possible to find an ELF-style encoding, especially for more exotic consequence relations. It should, we suspect, be straightforward to directly describe the inference rules and proofs of a very wide range of logics using ADTs. Second, unlike our rules, ELF-style proof-rules cannot be seen as specifying an implementation of the logic, except in an indirect way via an interpreter for proof construction in the framework logic. Finally, there is a significant difference in our approaches when it comes to metareasoning. Since we specify a single inductively constructed proof type (as opposed to a family of proof types parameterized by Wffs), we can equip it with induction and computation rules which are necessary for most nontrivial kinds of metatheory.

With respect to encoding style, the use of the framework logic FS_0 (as proposed by Feferman in [18] and carried out by Matthews [31]) is closest to our approach. FS_0 is a theory of functions and classes of expressions embedded in second-order predicate calculus. The basic data-type is the S-expression and through this one defines by inductive definition the classes of formulas and proofs for desired logics. As with our approach, functions may be defined by primitive recursion (on S-expressions) and reasoned about by induction. FS_0 appears to be designed as a minimal logic for allowing such reasoning (it is a conservative extension of primitive recursive arithmetic). By necessity, one cannot mix the syntax of the encoded logic and the framework logic; all notions of syntax must be explicitly encoded using S-expressions. In practice, this does not appear to cause significant problems. In [32] an example is provided about how one can formalize and reason about bound variables in FS_0 in the context of a prenex normal form theorem for predicate calculus. The conclusion there, which we find comforting, is that this kind of formalization and reasoning is not as hard as is commonly assumed. It is worth noting that the FS_0 account could be given directly in type theories such as Nuprl using inductive types to define terms and formulas.

6.2 Categorical Logics

Category theory like type theory is a unifying general language. Computer scientists have also found it a suitable context for investigating the basic questions being addressed here, such as what is a logic, what is an implementation, and so forth [23, 33, 28]. Moreover category theory has been the language of choice for some of the pioneering work on how to combine and relate large structures such as theories [10, 11]. It is also the standard setting in which to frame the algebraic approach to data types [7].

Our use of ADT's to define logics shows a clear and simple connection to the study of algebraic data types in a general setting. The closest related work in categorical logic is Burstall and Goguen's abstract account of model theory as *institutions* [23] and Meseguer's abstract account of proof calculi and general logics [33]. Our approach has concentrated on proof theory, but the generality of type theory suggests ways of expressing many of the concepts from institutions. In relation to the categorical account of proofs, we can say that our approach is more concrete, for example our notion of an LJ proof is far more specific than Meseguer's notion of an *effective proof subcalculus*. We have not tried to say what a proof calculus would be in general.

We are not aware of work in categorical logic which is directed at generalizing or supporting metamathematics. But the formulation of logic in categorical terms may help us see how to generalize our account of metamathematics.

6.3 Metatheory

Cornell work Our approach to formalized metareasoning bears closest resemblance to the research program on constructive metatheory carried out at Cornell. In [13], Constable and Howe demonstrate how one can synthesize tactics, decision procedures, and the like, through theorem proving in the Nuprl system. In [24], Howe describes a substantial implementation of this kind of approach in which theorems were proved about representations of Nuprl terms (e.g., that one could rewrite and normalize such terms). Moreover, Howe's development included a means of associating representations of Nuprl terms, with Nuprl terms themselves, so his verified tactics and procedures could be applied to construct Nuprl proofs.

Another line of research was that of Constable and Knoblock [27] in which they formalized the structure of Nuprl proofs within an extension of Nuprl, called Nuprl^1 which contained data types for Nuprl terms and proofs. So again one could prove theorems about proofs and construct tactics by theorem proving. They, in fact, defined a hierarchy of metalogics, Nuprl^i each one the object logic of the one above it. Recent work [2] has refined this technique, providing a single proof type (as opposed to a hierarchy of proof types) within Nuprl that refers to itself and can be formally reasoned about within the Nuprl system.

The kind of metatheory that we propose in this paper is similar in spirit to the previous Cornell research, but differs in several significant respects. First, the Cornell work has been concerned with encoding the terms and proofs of a specific logic (e.g., Nuprl) with specific types, and developing metatheory over these types. Here we are suggesting that this approach generalizes and can be used for a wide variety of logics. Moreover, we propose that this can be done in an abstract way (using ADTs) that does not commit the logic encoder to choose a specific representation in the framework logic. Of course, the downside of this abstraction is that we cannot directly compute over our representations of terms and proofs as can be done with the more direct Nuprl encodings.

A second important difference is that the Cornell work has been concerned with using such encodings by means of some kind of reflective mechanism. That is, to use the results of their metatheory, a connection must be made between representations of Nuprl terms and proofs (and statements proved about them) and actual Nuprl syntax. Here we suggest that all theorem proving can be done at one level: the metalevel. Although our encodings can be seen as specifying an implementation of that logic, no implementation ever need be constructed; instead all reasoning (about provability, derived rules, etc.) can take place within the framework logic. Hence, reflection is not central to our work since there is only one theory, the framework theory.

It is of course possible to propose that the framework logic be reflective (indeed if one were using Nuprl it would be). The advantages of using reflection in any practical formalization is one of the themes of the Cornell work, but we are not considering these practical concerns here. Moreover, it is interesting to consider an ADT account of object theories which are reflexive. We have not carried this out, but it is a line of inquiry which is in the spirit of this paper. The significant point is that one might be able to reduce the amount of detail needed in any reflexive axiomatization by separating the specification from its many possible implementations.

FOL approach A quite different approach to metatheory has been undertaken in the FOL system [43] and its ancestors. FOL is a first-order logic that supports multiple logical theories. In particular, one such theory may be a metatheory about first-order logic in which theorems may be proven about representations of first-order formula. One moves between these theories using a reflection principle. In the original FOL system, metalevel axioms were simply asserted; however the system has been redesigned (and renamed as GETFOL) in a way that these can be generated in a sound manner from an analysis of the implementation [22, 6]. Whereas for us, the metatheoretic encoding is primary, the GETFOL approach suggests that, at least in some instances, one can take a complementary view and generate a metatheory from an implementation coded in a certain way. The GETFOL work is ongoing, however, and it remains to be seen how general the GETFOL methodology is.

A Appendix: Predicate Calculus

In this appendix we give a brief account of how terms, formulas, and proof rules that involve side conditions on variables may be formalized within the ADT setting. The account we give here is fairly standard and closely follows [20, 42].

We begin by defining the terms of the predicate calculus which are built from variables and function applications. Function names are specified by an ADT FNAME that has an associated arity function and a decision procedure for equality of names. 0-ary functions are constants.

$$\begin{array}{l} FNAME \mathrel{\hat{=}} \\ \quad\quad Fname : Type \\ \quad \# \ Arity : Fname \to Nat \\ \quad \# \ \forall x, y{:}Fname.\, x = y \vee x \neq y \end{array}$$

Note that we don't insist that members of *Fname* are enumerable. This could be added if required in the metatheory. The ADT of terms may now be given.

$$\begin{array}{l} TERM \mathrel{\hat{=}} fn{:}FNAME \to \\ \quad\quad Term : Type \\ \quad \# \ Var : Atom \to Term \\ \quad \# \ Func : f{:}Fname(fn) \times NTuple(Arity(f), Term) \to Term \end{array}$$

$NTuple(n, T)$ is the type of n-tuples with members from the type T.

To define Predicate Calculus Wffs we need an ADT of predicate names, PNAME, which is defined similarly to FNAME. Wffs are parameterized by predicate names and the $Term$ type.

$$
\begin{aligned}
&WFF \mathrel{\hat{=}} f{:}FNAME \to pn{:}PNAME \to t{:}TERM \to \\
&\quad \mathit{Wff}: Type \\
&\#\ AWff:\ p{:}Pname(pn) \times NTuple(Arity(p), Term(t)) \to \mathit{Wff} \\
&\#\ IMP:\ \mathit{Wff} \times \mathit{Wff} \to \mathit{Wff} \\
&\#\ ALL:\ Atom \times \mathit{Wff} \to \mathit{Wff} \\
&\textit{...other connectives defined similarly}
\end{aligned}
$$

To define proof rules for terms of the predicate calculus we must formalize the concept of free variables, eigenvariables, and substitution. All of these may easily be defined by recursion on $Term$ and Wff using the induction principles these ADTs are equipped with. We use considerable syntactic sugar in the definitions that follow.

Let $SET(Atom)$ be the ADT of sets parameterized by the type of variables $Atom$ with carrier Set. Define FV_T on terms as follows.

$$
\begin{aligned}
FV_T(x) &= \{x\} \\
FV_T(f(t_1, ...t_n)) &= FV_T(t_1) \cup ... \cup FV_T(t_n)
\end{aligned}
$$

where the arity of f is n. Similarly define FV on Wffs by the following equations.

$$
\begin{aligned}
FV(p(t_1, ..., t_n)) &= FV_T(t_1) \cup ... \cup FV_T(t_n) \\
FV(p_1 \to p_2)) &= FV(p_1) \cup FV(p_2) \\
FV(\forall a.\, p_1) &= FV(p_1) - \{a\}
\end{aligned}
$$

Substitution must be defined both on formulas on terms. On terms, we define a substitution function which we shall display as $t_1[t_2/x]_T$.

$$
\begin{aligned}
y[t/x]_T &= \textit{if } y = x \textit{ then } t \textit{ else } y \\
f(t_1, ..., t_n)[t/x]_T &= f(t_1[t/x]_T, ..., t_n[t/x]_T)
\end{aligned}
$$

Substitution on formulas will be displayed as $f[t/x]$ and is as follows.

$$
\begin{aligned}
P(t_1, ..., t_n)[t/x] &= P(t_1[t/x]_T, ..., t_n[t/x]_T) \\
(p_1 \to p_2)[t/x] &= p_1[t/x] \to p_2[t/x] \\
(\forall y.B)[t/x] &= \textit{if } x = y \textit{ then } \forall y.B \textit{ else } \forall y.(B[t/x])
\end{aligned}
$$

We say that a term t is *free for* a variable x *in* a Wff p when when t can be substituted for occurrences of x in a capture avoiding way. We define the propositional function $FreeFor$ as follows.

$$
\begin{aligned}
FreeFor(t, x, p(t_1, ..., t_n)) &= TRUE \quad \text{(For } TRUE \text{ an non-empty type)} \\
FreeFor(t, x, p_1 \to p_2) &= FreeFor(t, x, p_1) \wedge FreeFor(t, x, p_2) \\
FreeFor(t, x, \forall y.p) &= x = y \vee x \neq y \wedge \neg(y \in FV(t)) \wedge FreeFor(t, x, p)
\end{aligned}
$$

In addition, to formalize proof rules we will need to know when a variable can be serve as an eigenvariable; that is, when it does not occur free among the hypotheses $\Gamma = [h_1, ..., h_n]$ of a sequent.

$$EigenVar(\Gamma, x) = \neg y \in (FV(h_1) \cup ... \cup FV(h_n)).$$

We conclude by indicating how a proof-type may be formalized for intuitionistic predicate calculus. This can easily be accomplished by slight modification to the type LJ_PROOF given in Figure 3.2. Specifically, parameterize this type over the newly defined Wff type instead of the type of propositional Wffs. The rules of propositional calculus hold in predicate calculus so they carry over directly. We must add rules for quantifier introduction and elimination such as the following.

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x.P}\, AllI \qquad \frac{\Gamma, P[t/x] \vdash G}{\Gamma, \forall x.P \vdash G}\, AllE$$

Both rules have side-conditions. In the first, we insist that $EigenVar(\Gamma, x)$ holds. In the second, we must have that $FreeFor(t, x, P)$ holds. Adding these rules is straightforward. We extend our proof type with the following two conjuncts.

$$\begin{array}{l}
\#\ AllI:\ x\!:\!Atom \\
\qquad \to p_1\!:\!\{p\!:\!LJ|EigenVar(Hyps(Seq(p)), x)\} \\
\qquad \to \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1)) \wedge Concl(Seq(p)) = All(x, Concl(Seq(p_1)))\} \\
\#\ AllE:\ w_1\!:\!Wff \\
\qquad \to w_2\!:\!\{w\!:\!Wff|IsAll(w) \wedge FreeFor(w_1, AllVar(w), (AllBody(w))\} \\
\qquad \to p_1\!:\!\{p\!:\!LJ|AllBody(w_2[w_1/AllVar(w_2)]) \in Hyps(p_1)\} \\
\qquad \to \{p\!:\!LJ|Hyps(Seq(p)) = Hyps(Seq(p_1) - AllBody(w_2[w_1/AllVar(w_2)]) + w_2 \\
\qquad\qquad\quad \wedge\ Concl(Seq(p)) = Concl(Seq(p_1))\}
\end{array}$$

Here, the auxiliary function $IsAll$ is true when its argument is a universally quantified Wff. $AllVar$ and $AllBody$ decompose $\forall x.P$ into the binding variable x and the body P.

Note that there are alternative ways to formalize these rules. In particular $\forall$-introduction is often formulated to allow the user to choose the name of the eigenvariable. Another option is that in $\forall$-elimination, rather than restricting w_2 to instances where $FreeFor$ is true, one may appropriately rename bound variables so that the substitution is made in a capture avoiding way.

Acknowledgements

We gratefully acknowledge discussions with Nax Mendler, Alan Smaill, and Sean Matthews. Frank Pfenning provided helpful comments on a draft of this paper.

References

[1] Peter Aczel. Term Declaration Logic and Generalized Composita *Proceedings of the Symposium on Logic in Computer Science.* IEEE, Washington, DC, 1991.

[2] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflect proof. In *Symposium on Logic in Computer Science*, Computer Society Press of the IEEE, 1990.

[3] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.

[4] Jon Barwise. Axioms for abstract model theory. *Ann. Math. Logic*, 7:221-265, 1974.

[5] Jon Barwise and Solomon Feferman (editors). *Model-Theoretic Logics.* Springer-Verlag, NY, 1985.

[6] David Basin, Fausto Giunchiglia, and Paolo Traverso. Automating meta-theory creation and system extension. In *AI*IA-91 (Italian Association for Artificial Intelligence*, Palermo, Italy, 1991. To Appear.

[7] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In *Automata, Languages and Programming. Lecture Notes in Computer Science, vol 81*, Springer-Verlag, NY, 1980, 76-90.

[8] N.G. de Bruijn. A survey of the project Automath. In *To H.B. Curry: Essays in combinatory logic, lambda calculus and formalism.* Academic Press, Reading, 1980,579-606.

[9] Rod Burstall. *Computer Assisted Proof for Mathematics: an Introduction Using the LEGO Proof System.* Technical Report ECS-LFCS-91-132, University of Edinburgh, 1991.

[10] Rod M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *5th IJCAI*, pages 1045 – 1058, Boston, Mass, 1977.

[11] Rod M. Burstall and Joseph A. Goguen. The semantics of clear, a specification language. In *Advanced Course on Abstract Software Specification*, LNCS 86, 1980.

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

[13] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In *A source book of formal approaches to A.I.*, North Holland, 1990.

[14] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 95–120, 1988.

[15] Albert G. Dragalin. *Mathematical Intuitionism – introduction to proof theory* Translations of mathematical monographs vol. 67. American Mathematical Society, Providence, RI, 1988.

[16] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. Unpublished Manuscript (submitted to JSL).

[17] Roy Dyckhoff, Neil Leslie, and Stephen Read. MALT st andrews (maclogic – a proof assistant for first-order logic. *Computerised Logic Teaching Bulletin*, 2(1):51–60, 1989.

[18] Solomon Feferman. Finitary inductively presented logics. In *Logic Colloquim '88*, North-Holland, 1988.

[19] Solomon Feferman. Inductively presented systems and the formalization of meta-mathematics. In D. van Dalen *et al*, editor, *Logic Colloquium '80*, pages 95–128, North-Holland, Amsterdam, 1980.

[20] Jean H. Gallier. *Logic for Computer Science.* Harper & Row, 1986.

[21] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen.* North-Holland, Amsterdam, 1969.

[22] Fausto Giunchiglia and Paolo Traverso. Reflective reasoning with and between a declarative metatheory and the implementation code. In *IJCAI '91*, 1991.

[23] Joseph Goguen and Rod Burstall. Institutions: Abstract Model Theory for Computer Science. Technical Report CSLI-85-30 Center for the Study of Language and Information, Stanford University, Palo Alto, CA, 1985.

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

[25] Douglas J. Howe. Computational metatheory in Nuprl. In *9th International Conference On Automated Deduction*, pages 238–257, Argonne, Illinois, 1988.

[26] Douglas J. Howe. Equality in lazy lambda calculi In *Proc. of the Fourth Annual Symp. on Logic in Computer Science*, IEEE.,1990.

[27] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Annual Symp. on Logic in Computer Science*, IEEE., 1986.

[28] Joachim Lambek and Philip J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[29] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23:281-296, 1969.

[30] David B. MacQueen. Using dependent types to express modular structure. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM, SIGACT, SIGPLAN, 1986.

[31] Sean Matthews. *Meta-Level and Reflexive Extension in Mechanical Theorem Proving.* PhD thesis, University of Edinburgh, DAI, 1991. Expected.

[32] Sean Matthews, David Basin, and Alan Smaill. *Experience with FS_0 as a Framework Theory.* Technical Report, University of Edinburgh, DAI, 1991. In preparation.

[33] Jose Meseguer. General Logics. In H.D. Ebbinghaus *et al.*(editors), *Proceedings of Logic Colloquium 87.* North-Holland, 1989.

[34] John C. Mitchell. Representation independence and data abstraction. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, ACM, SIGACT, SIGPLAN, 1986.

[35] John C. Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[36] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, ACM, SIGACT, SIGPLAN, 1985.

[37] Christine Paulin-Mohring. Inductive definitions in the calculus of constructions. In *The Calculus of Constructions, Documentation and users's guide*, Project Formel, 1989.

[38] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[39] Don Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165-210,1988.

[40] Don Sannella and Andrzej Tarlecki. *Toward formal development of ML programs: foundations and methodology.* Technical Report 71, Laboratory for Foundations of Computer Science, 1989.

[41] Paul Taylor. *Using Constructions as a MetaLanguage.* Technical Report ECS-LFCS-88-70, University of Edinburgh, 1988.

[42] D. van Dalen. *Logic and Structure.* Springer-Verlag, 1980.

[43] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

A Canonical Calculus of Residuals

Yves Bertot †
INRIA Sophia Antipolis
06561 Valbonne Cedex
France

July 1991

1 Introduction.

This work studies the notions of *descendance* and *residuals*. These notions are interesting for proof development systems in two respects. First, they are an important tool in the study of reduction systems, especially for comparing evaluation strategies and studying optimal evaluation [1, 5, 8, 12, 13]. As such, they have a natural place in any toolbox for reasoning about and proving properties of programming languages. Second, proof development systems often use rewriting themselves, and residuals can be used efficiently in the "debugging" of tactics, since tactics are very analogous to programs and residuals have already been shown to be useful in debugging programs, mostly in parallel programming languages, that are also based on rewriting [3].

Most presentations use a simple labeling of terms to describe descendance and residuals. For any reduction system, one simply produces a labeled version of the system, together with a procedure to transform a derivation on labeled terms in a derivation on unlabeled terms (called *erasure*) and a procedure to transform a derivation on unlabeled terms in a derivation on labeled terms, given a labeling of the initial term (called *lifting*). Intuitively, a node or a position in the resulting term *descends* from a node or a position in the initial term through a unlabeled derivation D if they share the same label in any labeled reduction obtained by lifting D, if the nodes of the initial terms are all labeled with distinct labels.

The use of labels to describe this notion of descendance is rather contrived. First of all, the labels have no importance as such, only an infinite set of distinct labels is required. Second, lifting an unlabeled derivation to a labeled one is a sequential process. One has to start with a labeled term, compute the labels for the term obtained after the first reduction, and continue recursively with the rest of the derivation. This does not take advantage of the inherent transitivity of the descendance relation: if D and D' are two derivations such that the final term of D is the initial term of D', if position A descends from position B through derivation D' and position B descends from position C through derivation D then obviously A descends from C through the derivation that performs D then D'. The derivations D and D' need not be elementary.

We rely on the property of descendance that any position in the final term of a derivation always descends from at most one position in the initial term. We prefer to consider the

†bertot@sophia.inria.fr

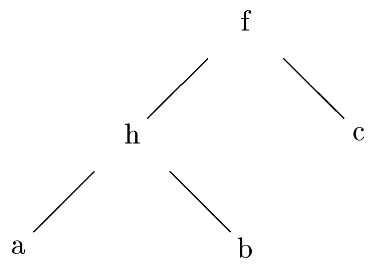

Figure 8: Graphical representation of the tree $f(h(a,b),c)$

converse of the descendance relation, which is a function and which we call an *origin* function. In what follows, we present a data structure for representing origin functions that is concise and well adapted to these reduction systems, together with an interpretation method based on a canonical rewriting system. We give the definition and the main properties of origin functions for reduction systems such as the λ-calculus and term rewriting systems. We show that these origin functions, together with the language we use to represent them, are a suitable alternative to labeling for implementing the notion of descendance.

2 Mathematical Preliminaries.

2.1 Occurrences.

We denote $\mathcal{N}$ the set of natural numbers and $\mathcal{N}_+$ the set $\mathcal{N} - \{0\}$. A *graded alphabet* is a set Σ given with a function $\alpha : \Sigma \to \mathcal{N}$, called *arity* function. The set $T(\Sigma)$ of Σ-*terms* is the least set verifying the following conditions:

1. For every operator $g \in \Sigma$ such that $\alpha(g) = 0$, one has $g \in T(\Sigma)$,

2. For every operator $f \in \Sigma$ such that $\alpha(g) = n > 0$ and every n-tuple $(t_1, \ldots, t_n)$, one has $f(t_1, \ldots, t_n) \in T(\Sigma)$.

The operator f is called the *head operator* of the term $f(t_1, \ldots, t_n)$. In the following Σ-terms are regularly called terms, when the alphabet is obvious from the context.

Finite trees are a suitable graphical representation for terms, as shown in Figure 1. This representation also hints that an alternative presentation of terms is possible, defining first the drawing with all the lines (the tree domain) and then labelling the nodes with operators. This kind of presentation is given in [7, 9].

For any number $i \in \mathcal{N}_+$, we denote s_i the function that maps any term $f(t_1, \ldots, t_n)$ such that $n \geq i$ to its i^{th} child, t_i, and is undefined for any term $f(t_1, \ldots, t_n)$ such that $n < i$. We denote $\mathcal{O}$ the set that contains the functions obtained by composing the function s_i and the identity of $T(\Sigma)$, *id*. We claim that the elements of $\mathcal{O}$ represent correctly *occurrences*, as they are presented in [7, 9]. The usual presentation is to consider occurrences as strings of natural numbers. Our argument is that the functions s_i represent the elementary strings of one integer and that composing functions represents concatenating strings. We simply denote $u(M)$ the subterm at occurrence u instead of denoting M/u. For this reason, we call *occurrences* the elements of $\mathcal{O}$. We denote $\circ$ the composition operator, with the semantics that $v \circ u(M) = v(u(M))$. For any two occurrences u and v, we note $u \leq v$ if there exists an occurrence w such that $v = w \circ u$, and we say that u is a *prefix* of v (because u is applied first).

A *tree domain*, D, is a subset of $\mathcal{O}$ that verifies the following properties:

1. For any two occurrence u and v such that $u \leq v$, if one has $v \in D$ then one has $u \in \mathcal{D}$,

2. For any two indices i and j such that $i \leq j$ and for any occurrence u, if one has $s_j \circ u \in D$ then one has $s_i \circ u \in D$.

In the following we consider only *finite* tree domains and simply call them tree domains. For any term M, we denote $\mathcal{D}(M)$ the set of occurrences that are valid on this term, this set is a tree domain, which we call the *domain* of M.

2.2 Reduction Systems.

In the following we consider a reduction system represented by the binary relation $\rightarrow$, and we write $M \rightarrow N$ to express that the term M reduces to N. However, this notation is not always sufficient since it does not give enough information on the manipulations that permit to go from M to N. For example, consider a term rewriting system that contains the following rule:

$$r: \qquad I(x) \rightarrow x$$

The reduction $I(I(a)) \rightarrow I(a)$ is a licit reduction for this system. But there are two ways to go from M to N: the first is to consider that $I(x)$ matches $I(I(a))$, thus performing the rewriting at the top of the term; the other is to consider that $I(x)$ matches $I(a)$, thus performing the reduction on the first subterm of $I(I(a))$. The two reductions are usually considered to be different (and they are for the notion of descendance), and we need extra information to decide which reduction is actually performed. We thus write $I(I(a)) \stackrel{(id,r)}{\rightarrow} I(a)$ to denote the first reduction, expressing that a rewriting is performed applying rule r at the occurrence id on the initial term. The extra information we add on the arrow need not always be the couple of an occurrence and a rule. In the $\lambda\beta$-calculus, we only need an occurrence, since we know that the rule is always the β-rule.

Computations in a reduction system are represented by the reflexive transitive closure of the reduction relation, noted $\twoheadrightarrow$, and the elements of this relation are called *derivations*. Here again, we may need extra information to specify exactly a derivation, and we write $M \stackrel{S}{\twoheadrightarrow} N$, where S is a list whose elements specify exactly all the elementary reductions that compose this derivation. For tree rewriting systems, S is a list of couples (*occurrence, rule*).

Since derivations are elements of a transitive closure, they can be concatenated. If $D = M \stackrel{S}{\twoheadrightarrow} N$ and $D' = N \stackrel{S'}{\twoheadrightarrow} P$ are two valid derivations in a reduction system, then $M \stackrel{S;S'}{\twoheadrightarrow} P$, there exists a derivation that goes from M to P by first performing the steps specified by S, then the steps specified by S', which we denote $D; D'$. For any reduction system $\mathcal{S}$, we denote $Deriv(\mathcal{S})$ the set of all valid derivations in this system.

2.3 Origin Functions.

The intuition in a reduction $M \rightarrow N$ is that some parts of the result N descend from parts of M and some parts of N may have been created during the reduction. Origin functions are introduced to describe this notion of descendance.

For any reduction system $\mathcal{S}$, an *origin mapping* $\Omega : Deriv(\mathcal{S}) \to (\mathcal{O} \to \mathcal{O})$ is a function verifying the following constraints:

1. If $D = M \xrightarrow{S} N$ is a derivation in $Deriv(\mathcal{S})$ and $f = \Omega(D)$, then one has $dom(f) \in \mathcal{D}(N)$ and $codom(f) \in \mathcal{D}(M)$. This constraint is not very important, but it states that the function f expresses the origins of the nodes in N and that these origins are all in M.

2. If D and D' are derivations in $Deriv(\mathcal{S})$ such that $D; D'$ is well defined, then one has $\Omega(D; D') = \Omega(D) \circ \Omega(D')$. This constraint expresses the intuition that the node at occurrence u descends from the node at occurrence w through the derivation $D; D'$ as soon as the node at occurrence u descends from the node at occurrence v by the derivation D' and the node at v descends from the node at w by D.

The images of derivations by Ω are called *origin functions.*

3 Representing Origin Functions.

3.1 Origins for Projections.

Occurrences act on terms like projections and they may receive origin functions. When we write $N = u(M)$, we may express that N is obtained by taking the subterm at occurrence u of M. Let f be the origin function associated to this manipulation. This origin function is very regular. The node at occurrence v in N descends from the node at occurrence v in $u(M)$, that is, the node at occurrence $v \circ u$ in M. If f is the origin function associated to this reduction, we have $f(v) = v \circ u$ for any occurrence $v \in \mathcal{D}(N)$.

We introduce a mapping $\mu : \mathcal{O} \to (\mathcal{O} \to \mathcal{O})$ that summarize this phenomenon. The mapping μ is defined by the following property:

$$\forall u, v \in \mathcal{O} \quad \mu(u) = f \leftrightarrow f(v) = v \circ u$$

The mapping μ is a morphism from $\mathcal{O}$ to $(\mathcal{O} \to \mathcal{O})$. Actually, $\mu(id)$ is the identity of $(\mathcal{O} \to \mathcal{O})$. The two spaces have a composition operator, $\circ$. With respect to these operators we have the following property:

$$\forall u, v, w \in \mathcal{O} \quad \mu(u) \circ \mu(v)(w) = \mu(u)(\mu(v)(w)) = w \circ v \circ u = \mu(v \circ u)(w)$$

Thus, we have $\mu(u) \circ \mu(v) = \mu(v \circ u)$, μ is contravariant. Last, μ is injective, since we have $\mu(u)(id) = u$.

Notice that for any occurrence u, the function $\mu(u)$ is defined for every occurrence in $\mathcal{O}$, instead of being defined only on a finite domain as are normal origin functions. When we state that the origin function f of a tree manipulation is equal to $\mu(u)$ ($f = \mu(u)$), we actually express f has the same value as $\mu(u)$ for every occurrence where it is defined. We shall see later that the possibility to abstract the domain definition this way is actually very useful in an implementation of origin functions.

3.2 General Case.

All tree manipulations are not as regular as projections, and we want to describe origin functions of arbitrary complexity. However, we can take advantage of the regularity given by the fact that terms are recursively definable.

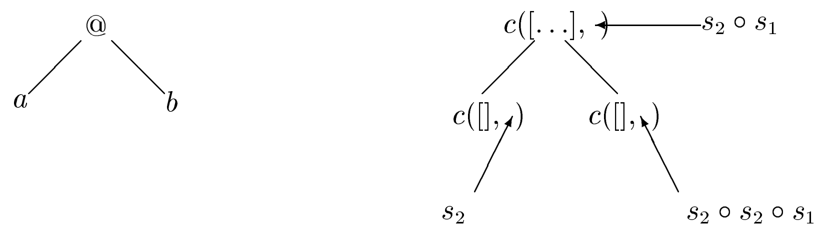

Figure 9: Graphical representation of origin functions.

When specifying the value of a term M, one only needs to provide an operator f and a tuple of terms $(t_1, \ldots, t_n)$ and say that $M = f(t_1, \ldots, t_n)$. The terms t_1 to t_n may in turn be defined by the same operation, recursively, until one reaches leaves. When providing origins for the nodes in M, one only needs to provide an origin for the head node (at occurrence id) and use origin functions for the subterms t_1 to t_n. These origin functions may in turn be defined by the same operation, recursively.

Denoting $[f_1, \ldots, f_n]$ a tuple of functions and $(\mathcal{O} \to \mathcal{O})^*$ the set of tuples of functions over occurrences, we introduce the mapping $c : ((\mathcal{O} \to \mathcal{O})^* \times \mathcal{O}) \to (\mathcal{O} \to \mathcal{O})$ that maps a tuple of functions over occurrences and an occurrence to a function over occurrences defined by the following property:

1. For any tuple $[f_1, \ldots, f_n] \in (\mathcal{O} \to \mathcal{O})^*$ and any occurrence u, $c([f_1, \ldots, f_n], u)(id) = u$.

2. For any tuple $[f_1, \ldots, f_n] \in (\mathcal{O} \to \mathcal{O})^*$, any occurrences u and v, and any index $i \leq n$, one has $c([f_1, \ldots, f_n], u)(v \circ s_i) = f_i(v)$. This mimics the property that $v \circ s_i(f(M_1, \ldots, M_n)) = v(M_i)$.

3. For any tuple $[f_1, \ldots, f_n] \in (\mathcal{O} \to \mathcal{O})^*$, any occurrences u and v, and any index $i > n$, one has $c([f_1, \ldots, f_n], u)(v \circ s_i)$ is undefined.

Example. *Let us consider the following reduction, taken from the λ-calculus (we use a term representation instead of the usual concrete syntax for λ-terms, to emphasize the tree structure).*

$$@(\lambda(x, @(x, b)), a) \to @(a, b)$$

The origin function for this reduction, f, is defined by the following equations:
$f(id) = s_2 \circ s_1$ $\quad$ $f(s_1) = s_2$ $\quad$ $f(s_2) = s_2 \circ s_2 \circ s_1$
The function f can also be represented by the following notation:

$$f = c([c([], s_2), c([], s_2 \circ s_2 \circ s_1)], s_2 \circ s_1)$$

This representation permits to represent extensively an origin function on a domain, without having to write the elements of the domain. We actually construct a tree isomorphic to the term whose origins are described, labeling each node with the origin for the corresponding occurrence, as is shown in figure 2. Since it is an extensive representation of the origin function, the notation using function c is powerful enough to represent any origin function.

The composition of two functions described using function c is also representable using this function. Actually, one only needs that the function on the right-hand side of the composition operator be represented using c. One obtains the following equality:

$$f \circ c([f_1, \ldots, f_n], u) \quad = \quad c([f \circ f_1, \ldots, f \circ f_n], f(u)) \tag{1}$$

This equality is a direct consequence of the definition of c, but it can be explained intuitively. Let us consider the final term of a manipulation is $P = op(t_1, \ldots, t_n)$, that each of the terms t_i is obtained from a term N by a manipulation whose origin function is f_i, that the origin of the head node in N is u, and that N is obtained from the initial term M by a manipulation whose origin function is f. Then each of the terms t_i is obtained from M by a manipulation whose origin function is $f \circ f_i$ and the origin of the head node of P in M is given by the origin of the node of N at occurrence u, $f(u)$.

The composition of a function represented using c and of a function represented using μ is also easy to describe, this time when the representation using c is on the left-hand side. Actually we have to use the fact that μ is composed of elementary occurrences s_i. One obtains the following equality:

$$c([f_1, \ldots, f_n], u) \circ \mu(v \circ s_i) \quad = \quad f_i \circ \mu(v) \tag{2}$$

Here again, this equality can be described intuitively. If P is obtained from N by taking the subterm at occurrence $v \circ s_i$, if $N = op(t_1, \ldots, t_n)$, and if the j^{th} child of N is obtained from the initial term M by a manipulation whose origin function is f_j, then P is obtained by taking the subterm at occurrence v in the term t_i, whose origins are described by the function f_i.

4 Representing Origins with Trees.

4.1 A Formal Language of Origins.

We introduce the language $Orig$ containing the two sorts $\mathcal{F}$ and $\mathcal{T}$ based on the following abstract syntax, where the index i ranges over $\mathcal{N}_+$:

$$\begin{aligned} \mathcal{F} &= s_i \mid id \mid \mathcal{F} \circ \mathcal{F} \mid c(\mathcal{T}, \mathcal{F}) \mid nil \\ \mathcal{T} &= [\mathcal{F}*] \end{aligned}$$

We call Occ the subset of $Orig$ that contains the trees constructed only with the operators s_i, id, and $\circ$. The operator nil is provided to represent undefined values.

Examples of trees in $Orig$ are $s_1 \circ s_2 \circ id$, $s_1 \circ nil$, and $c([s_1 \circ s_2, nil], c([], id))$.

The trees of Occ can be used to represent occurrences, using the interpretation function $\langle\!\langle . \rangle\!\rangle : Occ \rightarrow \mathcal{O}$ defined by the following properties:

1. $\langle\!\langle id \rangle\!\rangle = id$,
2. $\forall i \in \mathcal{N}_+ \quad \langle\!\langle s_i \rangle\!\rangle = s_i$,
3. $\langle\!\langle u \circ v \rangle\!\rangle = \langle\!\langle u \rangle\!\rangle \circ \langle\!\langle v \rangle\!\rangle$.

The trees of $Orig$ can be used to represent functions over occurrences, using the interpretation function $[\![.]\!] : Orig \rightarrow (\mathcal{O} \rightarrow \mathcal{O})$ defined by the following properties:

1. $[\![nil]\!] = nil$, where nil is the function that is undefined on any occurrence of $\mathcal{O}$.

2. $[\![id]\!] = 1_{\mathcal{O}}$,

3. $[\![s_i]\!] = \mu(s_i)$,

4. $[\![t_1 \circ t_2]\!] = [\![t_2]\!] \circ [\![t_1]\!]$,

5. $[\![c([t_1, \ldots, t_n], t)]\!] = c([[\![t_1]\!], \ldots, [\![t_n]\!]], [\![t]\!](id))$. Note that this function is undefined on id when $t = nil$.

Note that these trees differ from the notations we have used so far for origin functions in two respects. First, the composition operator takes its children in the reverse order, when compared to the composition of the functions the children represent. This happens because this composition operator is used in *Orig* to represent two kinds of compositions: the composition of occurrences and the composition of functions over occurrences. However, the two operations are related by the morphism μ, which is contravariant. This property must necessarily appear in one of the interpretations. The reader can check that this permits to have the following property:

$$\forall t \in Occ \quad [\![t]\!] = \mu(\langle\!\langle t \rangle\!\rangle) \tag{3}$$

Second, a tree with the head operator c accepts a tree of the same sort for second child, whereas the mathematical function c we use to describe origin function takes an occurrence as second argument, not an origin function. Here again, this property appears in the interpretation function, that forgets everything about the given origin function, except for the value in id. However, this property gives us the following equation:

$$\forall t \in Occ \quad [\![c([t_1, \ldots, t_n], t)]\!] = c([[\![t_1]\!], \ldots, [\![t_n]\!]], \langle\!\langle t \rangle\!\rangle) \tag{4}$$

4.2 Simplification and Computation.

To compute the value of the origin function $[\![t]\!]$ where t is a tree of *Orig* on an occurrence represented by a term u of *Occ*, we want an algorithm that takes u and t and returns a tree $v \in Occ$ such that $[\![t]\!](\langle\!\langle u \rangle\!\rangle) = \langle\!\langle v \rangle\!\rangle$. The procedure we propose is based on the following remarks.

a. In the following cases the value $[\![t]\!]$ is easy to compute:

  1. $t \in Occ \Rightarrow [\![t]\!](id) = \langle\!\langle t \rangle\!\rangle$ (equation 3),
  2. $t = c(M, t') \wedge t' \in Occ \Rightarrow [\![t]\!] = \langle\!\langle t' \rangle\!\rangle$ (equation 4),
  3. $t = nil \Rightarrow [\![t]\!]$ is undefined.
  4. $t = c(M, t) \wedge t' = nil \Rightarrow [\![t]\!]$ is undefined.

b. If one has $u \in Occ$ and $t \in Orig$, then the value $[\![t]\!](\langle\!\langle u \rangle\!\rangle)$ is equal to $[\![t']\!](id)$ where $t' = u \circ t$ (this is because $[\![u \circ t]\!](id) = [\![t]\!] \circ [\![u]\!](id)$ and $[\![u]\!](id) = u$).

Following these remarks, we provide a canonical rewriting system for which the normal form of any term in *Orig* matches one of the cases of remark a and such that the origin function represented by a term is preserved in any derivation. The procedure is then as follows, to compute $[\![t]\!](\langle\!\langle u \rangle\!\rangle)$ when $t \in Orig$ and $u \in Occ$:

1. normalize the term $u \circ t$ in a term t'.

2. If the head operator of t' is c then the second child of t' is a term t'' such that $[\![t]\!](\langle\!\langle u\rangle\!\rangle) = \langle\!\langle t''\rangle\!\rangle$, else $[\![t]\!](\langle\!\langle u\rangle\!\rangle) = \langle\!\langle t'\rangle\!\rangle$.

The rewriting system ρ that we propose is composed of the following rules:

$$\begin{array}{lrcl}
\rho_1 : & (t \circ t') \circ t'' & \rightarrow & t \circ (t' \circ t'') \\
\rho_2 : & s_i \circ c([t_1, \ldots, t_i, \ldots, t_n], t) & \rightarrow & t_i \\
\rho_3 : & c(T_1, c(T_2, t)) & \rightarrow & c(T_1, t) \\
\rho_4 : & c([t_1, \ldots, t_n], t) \circ t' & \rightarrow & c([t_1 \circ t', \ldots, t_n \circ t'], t \circ t') \\
\rho_5 : & id \circ t & \rightarrow & t \\
\rho_6 : & nil \circ t & \rightarrow & nil \\
\rho_7 : & t \circ nil & \rightarrow & nil \\
\rho_8 : & c([nil, \ldots, nil], nil) & \rightarrow & nil
\end{array}$$

Actually, rules ρ_2 and ρ_4 are rule schemas, corresponding to every possible combination of i and the length of the tuple between the square brackets $[\ldots]$.

The fact that the represented function is preserved in every reduction must be proved for every reduction. For rule ρ_1, it comes directly from the associativity of composition, for rule ρ_2 it commes from equation (2), for rule ρ_3 it comes directly from point 5 of the definition of $[\![\ldots]\!]$, for rule ρ_4 it comes from equation (1), for rule ρ_5 it comes directly from the fact that id represents the identity of $\mathcal{O}$, for rules ρ_6 and ρ_7 it comes from the definition of composition, and for rule it comes from the definition of c.

That ρ is terminating is proved by a simple ordering on terms using weight $P : Orig \rightarrow \mathcal{N}$ defined as follows:

- $P(id) = 2$,
- $P(s_i) = 2$,
- $P(t_1 \circ t_2) = P(t_1)^2 P(t_2)$,
- $P(c([t_1, \ldots, t_n], t) = \left(\sum_{1 \le i \le n} P(t_i)\right) + P(t)$.
- $P(nil) = 2$.

For any reduction $t \rightarrow t'$ using this rewriting system, one has $P(t') < P(t)$.

All the critical pairs in this rewriting system are confluent. This can be verified automatically using the algorithm of completion of Knuth-Bendix [11]. These properties are enough to prove that this rewriting system is confluent. To complete our study of ρ, we simply need the following property.

Property: *For any term t normal for ρ, one has:*

- $t = t_1 \circ t_2 \Rightarrow t \in Occ$,
- $t = c(M, t') \Rightarrow t' \in Occ \vee t' = nil$.

Proof. First remark that any subterm of a normal term is also normal. We can then prove these properties by induction. If $t = t_1 \circ t_2$, then the head operator of t_1 and t_2 cannot be $[\,]$ because of the syntax and it cannot be c or nil because t would not be normal. Then it is necessarily $\circ$, id, or s_i. If the term t_i has a head operator $\circ$ then the induction hypothesis yields $t_i \in Occ$, otherwise t_i is trivially in Occ.

Now, if t is normal and $t = c(M, t')$ then the head operator of t' cannot be $[\,]$ because of the syntax and it cannot be c because t would not be normal for ρ. Then we can apply the same reasoning for t' as above and obtain $t' \in Occ$ or $t' = nil$. $\square$

5 Applications.

5.1 Linear Term Rewriting Systems.

Term rewriting systems are usually described by rules of the form $\alpha \to \beta$ where α and β contain variables. The meaning attached to these rules is that any instance of α can be replaced by the corresponding instance of β. To express that a rewriting can be performed in any context, one simply adds a rule of the form:

$$\frac{t_i \to t_i'}{f(t_1, \ldots, t_i, \ldots, t_n) \to f(t_1, \ldots, t_i', \ldots, t_n)}$$

for any operator f in the language and any rank i. Conditional rewriting systems can be obtained by removing some of these rules as in [14], or by adding a predicate P that limits the applicability of the rule, yielding a rule of the form:

$$\frac{t_i \to t_i' \qquad P(f(t_1, \ldots, t_n))}{f(t_1, \ldots, t_i, \ldots, t_n) \to f(t_1, \ldots, t_i', \ldots, t_n)}$$

As shown by the Centaur experiment [4, 10], these rules can be directly transformed into executable code that allows the computation of derivations. An application of our work is to enhance this code, so that we compute the origin function of a derivation while computing this derivation.

We simply manipulate the rules that describe reductions, replacing uses of the formula $t \to t'$ (which means t reduces to t') by uses of the formula $t \to t', w$ (which means t reduces to t' with the origin function w), where the extra parameter w is written in the language $Orig$.

For conditional rules of the form above, the manipulation is systematic, yielding new rules of the following form:

$$\frac{t_i \to t_i', w \qquad P(f(t_1, \ldots, t_n))}{f(t_1, \ldots, t_i, \ldots, t_n) \to f(t_1, \ldots, t_i', \ldots, t_n), c([s_1, \ldots, w \circ s_i, \ldots s_n], id)}$$

The term added in the conclusion, $c([s_1, \ldots, w \circ s_i, \ldots s_n], id)$ is rather intuitive: the parts s_1, $\ldots$, s_j $(j \neq i)$, $\ldots$, s_n express that the children that are not affected by the reduction simply come from the corresponding children of the initial term. The part $w \circ s_i$ expresses that t_i' is obtained from t_i by a reduction whose origin function is w and t_i is obtained from the initial term by taking the i^{th} subterm. The part id indicates that the head node of the resulting term comes from the head node of the initial term.

Note that the use of the function μ in the interpretation $[\![.]\!]$ of trees of $Orig$ enables us to express that the origins for the subterm t_j are simply given by the data structure s_j. Without μ and the possibility it gives to use occurrences as functions over occurrences, it would be impossible to abstract over the variables t_j and we would be obliged to provide a distinct term constructed using $c([\ldots], .)$ for every possible instance of t'. In this sense, our data structure is particularly adapted to structural semantics.

For the axiomatic rules of the form $\alpha \to \beta$, we can also have a systematic treatment, if we restrict our manipulation to linear rewriting systems. Such systems verify the constraint that all the variables in β appear in α and that the variables appearing in α appear only once. For these rules the descendance relation is simple to express: any node that appears under a variable of β in the resulting term descends from the node that appears at the same place relative to the same variable of α in the initial term. This definition is not ambiguous because we are working with linear rewriting systems. This manipulation yields rules like the following one:

$$\mathit{double}(x) \to \mathit{sum}(x,x), c([s_1, s_1], \mathit{nil})$$

The term added to the reduction, $c([s_1, s_1], \mathit{nil})$ is simple to understand: both children of the resulting term descend from the only child of the initial term. The head node of the resulting term is created by the reduction, its origin is undefined and represented by nil.

5.2 The λ-calculus.

The λ-calculus is also described using conditional rules, but some of these rules are not based on rewrite formulas. The kernel of the λ-calculus is the β-rule:

$$\beta \qquad (\lambda x.M)N \;\to\; M[x\backslash N]$$

The notation $M[x\backslash N]$ represents the term M where every free instance of the variable x is replaced by N. This operation must also contain some renaming of the bound variables in M, so as not to catch free variables in N. This operation can be described by the following rules:

$$\mathit{subst} \qquad x[x\backslash T] \;=\; T$$

$$\mathit{diff} \qquad y[x\backslash T] \;=\; y \qquad (y \neq x)$$

$$\mathit{app} \qquad \frac{M[x\backslash T] = M' \qquad N[x\backslash T] = N'}{MN[x\backslash T] = M'N'}$$

$$\mathit{lambda} \qquad \frac{M[y\backslash z] = M' \qquad M'[x\backslash T] = M''}{\lambda y.M[x\backslash T] = \lambda z.M''} \qquad (z \text{ new})$$

To add computations over origin functions to these rules, we have to consider the whole notation $M[x\backslash N]$ to be a term, since the resulting term comes sometimes from N (rule *subst*) and sometimes from M (rule *diff*). In that case however, the β rule must be written as follows:

$$\beta \qquad \frac{M[x\backslash N] = M'}{(\lambda x.M)N \to M'}$$

Also, the rule *subst* does not look like a linear rule, since x appears twice in the initial term. It happens to be harmless, since this variable does not appear at all in the resulting term.

The language *Orig* is still adapted to represent origins in the substitution operation. The manipulation yields the following rules:

$$
\begin{array}{lrcl}
\textit{subst} & x[x\backslash T] & = & T, s_3 \\[1em]
\textit{diff} & y[x\backslash T] & = & y, s_1 \qquad (y \neq x)
\end{array}
$$

$$
\textit{app} \qquad \frac{M[x\backslash T] = M', w_1 \qquad N[x\backslash T] = N', w_2}{\begin{array}{l} MN[x\backslash T] = M'N', \\ \qquad c([w_1 \circ c([s_1 \circ s_1, s_2, s_3], id), w_2 \circ c([s_2 \circ s_1, s_2, s_3], id)], s_1) \end{array}}
$$

$$
\textit{lambda} \qquad \frac{M[y\backslash z] = M' \qquad M'[x\backslash T] = M'', w}{\lambda y.M[x\backslash T] = \lambda z.M'', c([s_1 \circ s_1, w \circ c([s_2 \circ s_1, s_2, s_3], id)], s_1)} \qquad (z \text{ new})
$$

Note how the rule *app* has been manipulated. The function w_1 is composed to a term that describes how $M[x\backslash T]$ descends from $MN[x\backslash T]$. The same kind of manipulation appears in the rule *lambda*. Note that in this rule the first premise, which only computes a variable renaming, is not used in the origin computation.

This manipulation of the rules for substitution goes along with a manipulation of the rule β that yields the following rule:

$$
\frac{M[x\backslash N] = M', w}{(\lambda x.M)N \rightarrow M', w \circ c([s_2 \circ s_1, s_1 \circ s_1, s_2], nil)}
$$

All these rule manipulations can actually be automatized, as is shown in [2].

5.3 Coming Back to Labeling.

Labeling a term M with letters from an *alphabet* $\mathcal{A}$ corresponds to providing a function $\mathcal{L} : \mathcal{D}(M) \rightarrow \mathcal{A}$ that maps any occurrence to the label carried by the corresponding node.

As we did for origin functions, we can define a function $c_{\mathcal{A}} : (\mathcal{O} \rightarrow \mathcal{A})* \rightarrow (\mathcal{O} \rightarrow \mathcal{A})$ that permits to deduce the labeling of a term from the labelings of its subterms. If $t_1, \ldots, t_n$ are terms labeled using the labeling function $\mathcal{L}_1, \ldots, \mathcal{L}_n$, if f is an operator and a is a label, then the term $f^a(M_1, \ldots, M_n)$ is actually labeled using the function $\mathcal{L} = c([\mathcal{L}_1, \ldots, \mathcal{L}_n], a)$ such that $\mathcal{L}(id) = a$ and $\mathcal{L}(v \circ s_i) = \mathcal{L}_i(v)$. In this respect, the origin functions simply correspond to labeling function where we have simply taken the set of occurrence $\mathcal{O}$ as labeling alphabet.

To compute the labeling $\mathcal{L}'$ of a term M' resulting from a derivation starting from a term M labeled with a labeling function $\mathcal{L}$, we can simply use the origin function of this derivation f. We simply have to write $\mathcal{L}' = \mathcal{L} \circ f$. A representation of $\mathcal{L}'$ can easily be constructed if we use $\mathcal{A}$ and the function $c_{\mathcal{A}}$ to define a language analogous to *Orig*.

6 Conclusion.

In this paper, we have presented a new paradigm for the study of descendance and labelings in reduction systems such as term rewriting systems and the λ-calculus. We have shown that this paradigm is especially suited for implementing this notion and we have described a data-structure and a method to compute using this data structure.

This method of computation consists of a canonical term rewriting system. The fact that we use a canonical term rewriting system has the main advantage that we do not have to specify the evaluation strategy. This permits to parallelize any computation on these data structures. If we consider a derivation $D = D_1; \cdots; D_n$, we can compute in parallel and independently the origin functions of the sub-derivations D_i and then compose these origin functions to obtain an origin function for the complete derivation. This is a big advantage over usual labelings, where it is arduous to compose labelings and the lifting of a derivation to a labeled one has to be done sequentially.

The methods used in this paper permit to describe the simple labelings that are related to descendance. However, more elaborate labelings have also been developed to study other aspects of reductions [6, 12, 14]. Those labelings do not use a simple alphabet, but rather algebras based on an alphabet. It is interesting to see how the operators of our language *Orig* and the operators of these algebras interact, in order to provide an implementation of these elaborated labelings.

References

[1] H. P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics* North-Holland, 1984.

[2] Y. Bertot, *Une automatisation du Calculs des Résidus en Sémantique Naturelle*, Ph.D Thesis, University of Nice, France, 1991.

[3] Y. Bertot, "Occurrences in Debugger Specifications", *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, 1991.

[4] P. Borras et al., "Centaur: the System", *Proceedings of the Third Symposium on Software Development Environments, ACM SIGSOFT'88*, Boston, Massachusetts, 1988.

[5] H. B. Curry, R. Feys, *Combinatory Logic, Vol. I*, North-Holland, 1958.

[6] J. Field, "On Lazyness and Optimality in Lambda Interpreters: Tools for Specification and Analysis", *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1990.

[7] J. H. Gallier, *Logic For Computer Science*, Harper & Row, 1986.

[8] G. Huet, J.-J. Lévy, "Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems", IRIA Laboria Report 359, 1979.

[9] G. Huet, "Deduction and Computation", *Fundamentals in Artificial Intelligence*, W. Bibel, Ph. Jorrand (editors), Springer Verlag LNCS 232, 1986.

[10] G. Kahn, "Natural Semantics", *Programming of Future Generation Computers*, K. Fuchi, M. Nivat (editors), North-Holland, 1988 (also appears as *INRIA Report no. 601*).

[11] D. Knuth, P. Bendix, "Simple Word Problems in Universal Algebras", *Computational Problems in Abstract Algebra*, J. Leech (editor), Pergamon, 1970.

[12] J.-J. Lévy, *Réductions Correctes et Optimales dans le Lambda Calcul*, Thèse de Doctorat d'état, University of Paris VII, France, 1978.

[13] J.-J. Lévy, "Optimal Reductions in the Lambda-Calculus", *to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin, J. R. Hindley (editors), Academic Press, 1980.

[14] L. Maranget, "Optimal Derivations in Weak Lambda-Calculi and in Orthogonal Term Rewriting Systems", *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, Orlando, Florida, 1991, pp. 255–269.

A semantics of evidence for classical arithmetic

Thierry Coquand*
Chalmers University

Preliminary version, June 1991

Introduction

This short note presents Gentzen's first proof of consistency of arithmetic, that was actually never published by Gentzen himself, but instead appeared first in a paper of Bernays [1] (but see also [3]). As emphasized by Bernays, this argument is easier to follow than the first published proof. It can be read directly as a game-theoretic analysis of the notion of classical truth: a formula is classically true iff there is a winning strategy for a game defined by this formula. This provides a semantics of evidence for classical first-order arithmetic (the term "semantics of evidence" seems due to B. Constable, see [2]). Furthermore, Gentzen's proof leads directly to the result that an existential statement provable in classical arithmetic is provable intuitionistically.

More importantly, when expressed game-theoretically, the dynamic aspect of cut-elimination becomes clearer. We believe indeed that the main object of study here is the analysis of the possible sequence of moves in the strategies corresponding to classical proofs.

Such an analysis suggests strongly that it should be possible to find a cut-elimination proof of a different nature than Gentzen's which reflects and is inspired by this dynamic aspect. We try to motivate this point by presenting such a proof for cuts of a low level of logical complexity, and by a conjecture expressing the termination of an internal communication, result that would refine Gentzen's cut-elimination.

We discuss next a concrete example, due to Gabriel Stolzenberg, which suggests that it can be computationally inefficient to break a multiple cut in its component. In the simplest possible case that departs from usual cut-elimination, we sketch a way to do this "multiple cut-elimination." Here also, it is directly checked that this "protocol for multiple cuts" works for cuts of low logical complexity.

At the end of the paper we present an inductive formulation of ω-logic, very close to Tait's formulation [9], which is readily seen to provide a computational content of classical arithmetical truth.

The contributions of this (preliminary) paper are:

- an "historical" contribution: we think it will be fair to attribute the result that an existential statement proved in Peano arithmetic has an intuitionistic proof at least partially to Gentzen, since this is a direct corollary of his first proof of normalisation [1],

*coquand@margaux.inria.fr

[1] we conjecture that it is because Gentzen's interpretation of a proof was in general a non deterministic algorithm that his first proof was for a while forgotten

- a formulation of a conjecture that refines Gentzen's cut-elimination, with a proof in a restricted case (that hopefully will be completed for the less preliminary version of this paper),
- the analysis of a concrete example where it is clear that the "multiple-conclusion" logic we manipulate cannot be simulated in a functional way, and so, the discovery of features of multiple-conclusion logic that are typical of parallel algorithms. We sketch then how to extend this to a truly parallel cut-elimination for classical arithmetic.

I would like to thank Gabriel Stolzenberg, Lars Hallnäs, Jan Smith, Peter Dybjer, Hugo Herbelin and Chet Murthy for enjoyable discussions on this topic. Karlis Cerans provided crucial critics.

1 A semantics of evidence

We start with a fixed language for arithmetic that contains (computable) functions like addition, multiplication and (decidable) basic relations, like equality, $\leq$, ... We suppose that whenever an atomic relation R is in this language we have another one R^* which represents its complement, in such a way that $(R^*)^*$ is R.

The formulae are built inductively form atomic formulae by conjunction &, disjunction $\vee$, universal and existential quantification. The negation φ^* is defined inductively from φ as usual.

To simplify things, we will suppose that all formulae are prenex formulae in which universal and existential quantifications alternate, that is of the form $\forall x \exists y \forall z \dots$, in which case we say that the formula is **universal**, or of the form $\exists x \forall y \exists z \dots$, in which case we say that the formula is **existential.** All quantifier free formulae are decidable.

1.1 The intuitionistic case

We recall first what is a possible game-theoretic semantics of evidence for intuitionistic logic, as presented for instance in A. Ranta's thesis [8]. We consider the following game between Nature and Myself, which consists in making **moves,** that are existential or universal instantiations, in a given formula φ, which is called the **configuration** of the game. Myself is trying to establish the truth of formula φ, and Nature tries to produce a counter-example. If the formula is atomic, then it is decidable: if it is false, Nature wins, otherwise, Myself wins. If the formula is of the form $\exists n\ A[n]$, Myself should produce an integer n_0 and the game goes on with $A[n_0]$. If the formula is of the form $\forall n\ A[n]$, Nature produces an integer n_0 and the game goes on with the formula $A[n_0]$.

For this game, a formula A can be defined to be intuitionistically true iff there is a winning strategy for Myself.

1.2 Extension to "multigames"

We can complicate this by allowing the configuration of the game to be a finite multiset of formulae. We write + the addition on multisets. The game stops when at least one formula is atomic and true, in which case Myself wins. In the other cases, Myself should make an instantiation whenever all formulae are existential, and Nature should make a move whenever at least one formula is universal, by instantiating the universal formulae. If all formulae are atomic and false, then Nature wins.

In this version, there is a winning strategy for Myself for the configuration of "excluded middle" $A + A^*$, for any formula A : Myself simply waits for Nature to move, and mimics her move in the dual formula.

For this notion of game however, it is not the case that there is a winning strategy for $\exists n \forall m\ [A[n]\ \vee\ A^*[m]]$ even in the case where A is decidable. Indeed, suppose that Myself has such a winning strategy. Myself has to give a value n_0 for n, because the formula is existential. We know that if $A[n_0]$ does not hold, then we have $\forall m\ A^*[m]$. Otherwise, Nature can win by playing m_0 such that $A[m_0]$ holds. By checking whether $A[n_0]$ holds or not, we would thus extract a decision algorithm for $\exists n\ A[n]\ \vee\ \forall m\ A^*[m]$.

Notice however that, as pointed out already, there is a winning strategy for the "equivalent" multiset formula $\exists n\ A[n] + \forall m\ A^*[m]$: Myself waits for an instantiation $m = m_0$ from Nature, and if $A^*[m_0]$ does not hold, win by playing $n = m_0$ (if $A^*[m_0]$ holds, then Myself wins already after Nature's move).

1.3 Games with "backtracking"

For getting a notion of game such that (intuitionistic) winning strategy contains classical provability, we allow backtracking for moves of Myself. This means that Myself can choose to complicate a configuration $M + \exists n\ A[n]$ where all the formulae are existential by both instantiating the formula $\exists n\ A[n]$ and keeping it, which produces the configuration $M + \exists n\ A[n] + A[n_0]$ where n_0 is the integer chosen by Myself. The moves of Nature are the same as before.

For this notion of game, there is a winning strategy for Myself for the configuration φ iff φ is classically true.

Instead of showing formally this equivalence, we will limit ourselves to show that if there is a winning strategy for the configuration $M + A + A$, then there is a winning strategy for the configuration $M+A$, and if there is a winning strategy for the configuration $M+A$ and a winning strategy for the configuration $N + A^*$, then there is a winning strategy for the configuration $M + N$.

This is enough to show that the notion of truth defined by the existence of a winning strategy has good properties. For instance, if we have a winning strategy for $N + A[0]$ and, for all n, a winning strategy for $M + A[n]^* + A[n+1]$, then we deduce from these two closure properties that there is a winning strategy for all n for $M + N + A[n]$.

The first claim is seen by simulating directly the moves of a strategy for $M + A + A$ by moves for the configuration $M + A$.

The second claim is more difficult, and we will present it as the proof of termination of some internal communications between two players following winning strategies.

1.4 Two examples

There is now for instance a winning strategy for $\exists n \forall m\ [A[n]\ \vee\ A^*[m]]$. Myself chooses any instantiation for n, for instance $n = 0$, and keeps the formula, waiting for a $m = m_0$ given by Nature. If $A^*[m_0]$, then Myself wins, and if $A[m_0]$ then Myself chooses $n = m_0$ for its next move.

Another example, which shows that we cannot bound a priori the number of backtracking in Myself's guess, is the following strategy for the statement

$$\exists n \forall m \; [f(n) \leq f(m)],$$

seeing f as an oracle. Myself starts by guessing any value for n, for instance $n = 0$, and allows himself to backtrack. Nature plays then $m = u_1$. If $f(0) \leq f(u_1)$, Myself wins. If $f(u_1) < f(0)$, Myself backtracks and plays $n = u_1$, and allows himself to backtrack. Nature plays then $m = u_2$. If $f(u_1) \leq f(u_2)$, Myself wins. Otherwise, Myself backtracks and plays $n = u_2$, and allows himself to backtrack, and so on.

This will stop eventually, because $<$ is well-founded, but it is not possible to bound a priori (without knowing anything about f) the number of times that Myself will have to backtrack.

This explanation of classical truth appears in Gentzen, see [3, 1]. Note that Bernays, in [1], presents this proof using choice sequences, for representing the sequence of moves of Nature. We can use inductive definitions instead to represent the notion of choice sequence, as done for instance in [6].

1.5 The case of existence statement

Let us look at the special case of a winning strategy for a configuration $\exists n \; A[n]$, where $A[n]$ is a (decidable) atomic formula. We see the decision procedure for $A[n]$ as an oracle. To have a winning strategy in this case means that Myself will do a finite number of wrong guesses for n, until he eventually finds a n_0 such that $A[n_0]$. We can actually suppose that Myself always is doing some "auto-censure" by himself, so that he checks internally whether or not his guess is correct for an existential formula $\exists n \; A[n]$, where A is atomic. With this assumption, a winning strategy for an existence statement is exactly a witness.

We thus get that the result "if an existence statement is provable in classical first-order arithmetic, then it is provable intuitionistically" follows from the identification of classical truth with the existence of a winning strategy.

1.6 Simple backtracking

In all the examples we have presented so far, the backtracking that Myself uses is of a particular nature. Myself never changes his mind about a value he has considered as wrong (we will precise this notion later). We call this behaviour of Myself **simple backtracking**.

This notion of simple backtracking is interesting because it does involve backtracking, but it is however a simple enough behaviour so that we can give a complete analysis of what is happening in the case where all players follow simple backtracking. In particular, we will be able to analyse later the case of multi-cuts for simple backtracking, that involves already real parallelism.

In order to analyse a little more this notion of simple backtracking, we introduce the following notations. In the history of configurations of a game which has $M + A$ as an initial configuration, where A is existential, we follow the moves in A by writing A_1, A_2, …the instantiations of A (due to Myself), and then A_{11}, A_{21}, …the respective instantiations of these instantiations (due to Nature), and so on. If a formula B is of the form $A_{n_1 \dots n_p}$, or $A^*_{n_1 \dots n_p}$, we say that $n_1 \dots n_p$ is the **index** of the formula B, and we write $n_1 \dots n_p = \mathsf{ind}(B)$. We say that a sequence $n_1 \dots n_p$ is a **direct extension** of a sequence $m_1 \dots m_q$ iff $p = q + 1$, and $n_i = n_i$ for $i < p$.

To say that the backtracking is **simple for** the formula A is to say that once we have observed the sequence of moves

$$A,\ A + A_1,\ A + A_{11},\ A + A_{11} + A_2, \dots, A + \dots + A_{n_1 \dots n_p}, \dots$$

with p even, then the next move of Myself will be $A + \dots + A_{m_1 \dots m_q}$, where $m_1 \dots m_q$ is strictly bigger than $n_1 \dots n_p$ for the alphabetical ordering on sequence of integers.

This is the case if the formula A is of the form $\exists x \forall y\ B[x, y]$, where B is atomic, simply because in this case, Myself cannot play A_{111}. Simple backtracking holds thus when we consider formulae of low logical complexity.

If Myself follows a strategy that uses only simple backtracking for an existential formula A, then we can represent Myself's moves in A (for a given game against nature) as a sequence of the form

$$A,\ A_1,\ A_{11},\ A_{111},\ A_{1111},\ A_{112},\ A_{1121},\ A_{113}, \dots$$

Such a sequence can be read as follows (if the sequence of quantifiers of A is $\exists x \forall y \exists z \forall t \dots$): Myself makes a first guess $x = x_1$ about the value of x. Nature then answers $y = y_{11}$. Myself persists in his choice by guessing $z = z_{111}$, this choice being refuted by Nature, who plays $t = t_{1111}$. At this point, Myself changes his mind about the choice of z (he considers that Nature has really refutated it and does not persist in his choice, leaving the last refutation of Nature t_{1111} without answers). He tries then $z = z_{112}$. This is refuted by Nature who plays $t = t_{1121}$.

One can well imagine a strategy where Myself changes his mind also about the fact that he was wrong. This is a more subtle kind of behaviour.

The analogy with learning theory, that Myself makes successive guesses according to the moves of Nature, seems clear here and should be precised. We will limit ourselves here to point out this analogy.

It would be nice if we can insure that for all classically true multiset M, it is possible to find a winning strategy using only simple backtracking. Though we don't have any concrete counter-example, we suspect that this is not the case, since we don't see how to get a strategy for the configuration $M + A$ from a strategy for the configuration $M + A + A$ if we impose strategies to use only simple backtracking. Any strategy for $M + A + A$ can be simulated by a strategy for $M + A$. Even if for both A, a strategy for $M + A + A$ uses only simple backtracking, this may not be the case for the simulated strategy for $M + A$.

Here is an example of a sequence of move that uses more than simple backtracking:

$$A,\ A + A_1,\ A + A_{11},\ A + A_{11} + A_2,$$

$$A + A_{11} + A_{21},\ A + A_{11} + A_{21} + A_{111},$$

$$A + A_{11} + A_{21} + A_{1111},\quad A + A_{11} + A_{21} + A_{1111} + A_{211}, \dots$$

The intuition is that Myself, when he plays A_{111}, changes his mind about his first backtracking, and does not answer to the refutation A_{21} of his last play. But when Nature refutes this last play A_{111} by playing A_{1111}, Myself changes his mind again and comes back to the choice of his second instantiation of A. Myself tries by playing A_{211} to refute the previous refutation of Nature A_{21} he had left without answers.

2 A dynamic view of cut-elimination

Let us imagine that we are playing against Nature for a given configuration. We can see a strategy for this configuration as a player we have at our disposition. Each time Nature plays, we transmit her move to this player. When all formulae are existential, we wait to see what is the move of this player on this configuration. We follow the strategy by copying his move.

With this picture in mind, we can conceive that it takes more or less time for the player to answer our question. We can consider a strategy to be **total** if we are sure that, eventually, after a finite amount of time, the player will answer. It is then natural to consider also **partial** strategies, that are like players who stay mute, thinking for a too long time about their next move.

Given two strategies for $M + A$ and $N + A^*$, we show how to build a "partial" strategy for $M + N$. The situation is exactly the same as in concurrency theory where partial processes appear when we use internal communication: a deadlock due to "infinite internal chatter" can happen when we combine these two strategies. But, and this is what the cut-elimination result expresses, if both strategies are winning strategies, then we do get a total strategy which is a winning strategy.

It seems possible to give a definition of this compound strategy by following Gentzen's argument. This corresponds to a direct study of the following property of a multiset of formulae M "there exists a winning strategy for M". One gives a direct inductive definition of this property, and one shows that if both $M + A$ and $N + A^*$ have this property, then so does $M + N$. The argument is a double induction, first on the complexity of the cut-formula, and then on the proof that the strategies are winning. Such a proof is presented in the last section. If we follow this approach however, it is not so clear what is the "structure" of the algorithm we get.

It seems much more interesting to see if a direct termination argument can be given, based on an analysis of what are the possible interactions between two players I and II.

2.1 Definition of the compound strategy

Here is an informal description of how this compound strategy is built. We assume that Myself has at his disposition two coplayers I and II. The player I represents a strategy for the game of configuration $M + A$ and the player II a strategy for the game of configuration $N + A^*$. Myself follows then the following protocol for the game against Nature of configuration $M + N$.

As long as one formula in M or N is universal, Myself waits for an instantiation coming from Nature. It transmits then this instantiation to the player I or II that is concerned with it.

After a finite number of such moves, both M and N have only existential formulae, so that Nature is waiting for an existential instantiation by Myself. Myself noticed that at least A or A^* is existential. Let us say that A is existential. In this case, Myself asks to the player I what is his move. Myself knows that I will answer, because all the formulae in the configuration of the player I are existential. There are two cases:

- I instantiates a formula in M, then Myself does the same move and the game goes on,
- I instantiates the formula A. This is the difficult case.

In the second difficult case, Myself has not yet available any play against Nature. Indeed, Nature does not "see" the formula A and so, cannot transmit I's move as in the first case.

There is a simple subcase however for which it is clear what Myself should do: if the player I instantiates the formula A without keeping it. In this case, the formula A becomes A_1, and Myself transmits this move to the player II. The configuration of I is then $M + A_1$, and the configuration of II is $N + A_1^*$. The situation is the same as before, except that Myself is now waiting for II's move. This "ping-pong" kind of play between I and II (via Myself) goes on for a finite amount of time, because the formula A is finite.

If during this internal communication, the formula A (resp. A^*) becomes true, then the dual formula becomes false, and the game goes on with only the player II (resp. I), since this player follows then a winning strategy for N (resp. M), the other player becoming inactive. If both players follow a winning strategy, it is then clear that Myself follows a winning strategy by copying one of these two players.

The only remaining subcase is when Myself gets to a position where, let say, I has a move in A, but this move is such that I keeps the formula A (allowing backtracking). In such a case, it is not so clear what Myself can do. Here is a possibility, that we shall analyse (and which corresponds to Gentzen's solution).

Myself transmits the move to the player II, the formula A^* becoming A_1^*, but also Myself *keeps a copy* of the player II in its initial configuration. The motivation is that Myself does not know whether or not the choice of the player I is definitive, and so, it makes sure that he can continue in case of a change of mind of the player I.

Once this is done, the configuration is almost like the previous subcase, except that Myself allows I to backtrack in his choice.

How is the game going on?? Myself asks to the latest copy of the player II, who plays now with a configuration $N + A_1^*$, what is his move. If this move is in N, it is transmitted to Nature. If this move is in A_1^*, Myself proceeds as before, i.e.:

- if this move is without backtracking (II is sure of his guess), then the configuration of II becomes $N + A_{11}^*$, and Myself transmits this move to the player I, whose configuration becomes $M + A + A_{11}$,
- if this move is with possible backtracking, then Myself does as before. The configuration of II becomes $N + A_1^* + A_{11}^*$, and Myself transmits this move to the player I, whose configuration becomes $M + A + A_{11}$, but Myself also keeps a copy of the player I in his configuration $M + A + A_1$, in case of a possible backtracking of II, and so on.

This finishes the description of the cut-elimination process.

2.2 Analysis of the problem of termination

The partial correctness of the compound strategy described in the previous subsection is clear. That is, if the players I and II follow a winning strategy for their respective configuration, it is clear that, if a game between Nature and Myself terminates, then Myself wins. However, it is not clear that the internal moves between the two players I and II terminate. In this subsection, we want to analyse the nature of the problem of termination.

First, it is not restrictive to assume that both players I and II, when they make a move in A, are doing an instantiation with possible backtracking. Second, we can suppose that there is no moves by I in M and no moves by II in N, so that we can restrict our attention to the moves in the formula A only. That is, we suppose that we have a (partial) strategy for A and a (partial) strategy for A^*, and we analyse what happens if we let the strategy for A play against the strategy for A^*. We can then analyse the possible "interaction paths": we are sure that we will observe the moves

$$A + A_1,\ A_1^* + A_{11}^*,$$

and for the next move, there is a choice: $A + A_{11}$ can become $A + A_{11} + A_{111}$ or $A + A_{11} + A_2$, and so on.

What we have to show is that if these moves come from winning strategies, then this interaction path is finite.

The problem appears then to be the following. We know that a winning strategy (for the player I) will win against any player that does not backtrack (and hence, in any such game, the player I will backtrack only a finite number of time). We want to generalise this to a play against a winning strategy (of II), which may backtrack. It is possible to show that if the strategy for II bacKtracks only a finite number of time, then the game will stop (because we have then a finitely branching tree, which has finite branches). The problem is that, a priori, the only way to be sure that this will happen is to show that I will backtrack a finite number of time. So there is a circularity here.

We can however formulate this problem as a pure problem of termination. The interaction can be represented as a sequence of formulae

$$B_0 = A^*,\ B_1 = A + A_1,\ B_2 = A_1^* + A_{11}^*,\ B_3 = A + A_{11} + A_2, \ldots$$

such that each B_k is a sum of formulae $C_1 + \ldots + C_n$ where

1. all C_i are existential for $i < n$, and C_n is universal, we call C_n the **end formula** of B_k, and we write $h(B_k) = C_n$, $t(B_k) = C_1 + \ldots + C_{n-1}$,

2. there is exactly one $i < n$ such that $\mathsf{ind}(C_n)$ is a direct extension of $\mathsf{ind}(C_i)$. Furthermore, $\mathsf{ind}(C_n)$ is the least direct extension of σ for the lexicographical ordering that does not occur already in $B_0, \ldots B_k$; we say then that i is **answered** in B_k,

3. for each $i < k$, the dual C_i^* appears as the end formula of exactly one B_j, for one $j < k$,

4. if i is answered in B_k, then $t(B_{k+1}) = t(B_i) + h(B_k)$.

The formulae $B_0, \ldots, B_k$ corresponds to the copies of the players I and II. We call such a sequence $B_0, \ldots, B_k$ an **interaction sequence.**

The refinement of Gentzen's cut-elimination is that such an interaction sequence is finite, if the players follow a winning strategy. Gentzen's argument (see the last section) provides only the existence of a winning strategy, without describing it. It does seem possible however to use a similar argument to show that the interaction sequence is finite, as will be shown in the next version of this paper.

2.3 Total correctness in some restricted cases

A termination argument is readily given if A is of low logical complexity, for instance of the form $\exists x\ B[x]$, or $\exists x \forall y\ B[x,y]$. This becomes more complicated in the case $\exists x \forall y \exists z\ B[x,y,z]$, and we leave to the reader an analysis of all possible behaviours of cut-elimination in this case.

We conjecture the termination of the algorithm in the general case. We will prove here only the case of simple backtracking.

With this added assumption, we use that A is finite, of depth n, and we remark that the number of formulae extension of $A_{n_1 \dots n_p}$ occurring in the interaction sequence is finite, by induction on $n-p$.

2.4 No-counter example interpretation

Another method (maybe inspired by Gentzen's proof) of giving a computational meaning of arithmetical truth is the no-counter example interpretation due to Kreisel. This is described for instance in the introduction of [6]. One can see the present "interaction approach" as an attempt towards the analysis of a computation at higher-type, in the spirit of Kleene 78 [5].

3 Towards a "symmetric" cut-elimination

We will analyse now a "multiple cut". We will limit ourselves to the case of a cut of the following form: we suppose to have a strategy for the games of configurations $M+A$, $N+A^*+B^*$ and $L+B$, and we try to build from these strategies a strategy for the game of configuration $M+N+L$.

It appears that there is only one choice if A or B is universal. If both A and B are existential however, there are several choices. The main point of this paper is to show that the usual two choices: first eliminate the cut between $M+A$ and $N+A^*+B^*$, then between $M+N+B^*$ and $L+B$, or the opposite solution, first eliminate the cut between $L+B$ and $N+A^*+B^*$, then between $L+N+A^*$ and $M+A$, not only lead to distinct results in general, but also are not natural w.r.t. the present game-theoretical analysis of cut-elimination.

3.1 A concrete example

This phenomena is particularly clear for the following concrete example, which is due to Gabriel Stolzenberg. It involves a typically classical lemma: the "infinite box principle".

The example is the following. We suppose given as an "oracle" a stream of 0 and 1. Classically, there are infinitely many 0, or infinitely many 1. The question is about a possible computational meaning of this assertion.

Let us represent the stream of 0 and 1 by a parameter f which is a unary function symbol. We represent a computational meaning of the infinite box principle as a winning strategy for the game of configuration

$$\forall x_0 \exists y_0\ [x_0 \le y_0\ \&\ f(y_0) = 0]\ + \forall x_1 \exists y_1\ [x_1 \le y_1\ \&\ f(y_1) = 1],$$

that we will write $A(0) + A(1)$. Here is such a winning strategy: Myself waits for two instantiations $x_0 = u_0$ and $x_1 = u_1$ from Nature. When Myself gets these values, he computes the value of $u = \max(u_0, u_1)$ and asks to the oracle the value of $f(u)$. If $f(u) = 0$ Myself wins by playing $y_0 = u$ and if $f(u) = 1$, Myself wins by playing $y_1 = u$.

This interpretation seems very natural.

The question now is to see how to use this "computational" interpretation to do some effective computation. Let us analyse a trivial use of this lemma: from the infinite box principle, we deduce that there are at least two 0s or at least two 1s. Indeed, if there are infinitely many 0, there are two 0s (by picking the first two 0s), and similarly in the other case.

We have so a winning strategy for the configuration

$$\exists a_0 \exists b_0\ [a_0 < b_0\ \&\ f(a_0) = f(b_0) = 0]\ + \exists x_0 \forall y_0\ [y_0 < x_0\ \vee\ f(y_0) = 1],$$

i.e. if there are infinitely many 0s, then there are two 0s. We will write $M(0) + A(0)^*$ this configuration. and a winning strategy for the configuration

$$\exists a_1 \exists b_1\ [a_1 < b_1\ \&\ f(a_1) = f(b_1) = 1]\ + \exists x_1 \forall y_1\ [y_1 < x_1\ \vee\ f(y_1) = 0].$$

i.e. if there are infinitely many 1s, then there are two 1s. We will write $M(1) + A(1)^*$ this configuration.

Let us precise this winning strategy for the configuration $M(0) + A(0)^*$. Myself starts by instantiating $x_0 = 0$, keeping the formula $A(0)^*$. Nature answers then by giving $y_0 = u$. If we have $u < 0\ \vee\ f(u) = 1$, then Myself wins. If we have $f(u) = 0$, then Myself backtracks in his choice of x_0, and choses now $x_0 = u + 1$. Nature answers by giving $y_0 = v$. If we have $v < u + 1\ \vee\ f(v) = 1$, then Myself wins. If we have $u + 1 \leq v$ and $f(v) = 0$, then Myself wins by playing $a_0 = u$ and $b_0 = v$.

We have now three winning strategies for the respective configurations $M(0) + A(0)$, $A(0)^* + A(1)^*$ and $A(1) + M(1)$, and we want a winning strategy for the configuration $M(0) + M(1)$. Notice indeed that such a winning strategy will provide us with two 0s or two 1s.

One way to solve this problem is to "put parenthesis". We analyse this way first.

3.2 "Usual" cut-elimination is not symmetric

It consists in reducing the problem of multiple cuts to the problem of simple cut. Symbolically, we want to do

$$\mathbf{CUT}(M(0) + A(0), A(0)^* + A(1)^*, A(1) + M(1))$$

and we do this by doing either

$$(*) \qquad \mathbf{CUT}(\mathbf{CUT}(M(0) + A(0), A(0)^* + A(1)^*), A(1) + M(1))$$

or by doing

$$(**) \qquad \mathbf{CUT}(M(0) + A(0), \mathbf{CUT}(A(0)^* + A(1)^*, A(1) + M(1))),$$

using the algorithm described above.

We will not do it explicitly here, leaving this as an exercise for the reader. The important points are that

- we *do not* get the same algorithm doing (*) and doing (**),
- both algorithms (*) and (**) are not symmetric w.r.t. 0/1, that is, they do not provide the same answer if we interchange 0 and 1 in the values of the oracle f.

3.3 A symmetric cut-elimination

It is remarkable that, on this simple example, a way of doing cut-elimination suggested by "common sense", which is neither (*) nor (**), furnishes an algorithm that is symmetric w.r.t. 0/1.

More generally, in the case of a cut between A, $A^* + B^*$ and B where A and B are existential, there seems to be a canonical way of doing the cut-elimination in the case of simple backtracking. This way differs from the one where we put parenthesis in general.

We will analyse this on one possible interaction sequence (which corresponds to one possible interaction sequence for the example described above).

Since A and B are existential, the corresponding strategies guess first values for them, with possible backtracking: A_1 and B_1. Then, Myself asks what is the move for $A_1^* + B_1^*$. Let say it is A_{11}^*. In this case, we consider that the choice A_1 has been refuted, and Myself transmits this refutation to the corresponding player. This player can either persist in his choice playing A_{111}, or changes his mind, playing A_2. In this last choice, since only simple backtracking is allowed, Myself can naturally consider that the choice A_1 has been definitively refuted and will never come back to this choice again. So, Myself asks what is the move for $A_2^* + B_1^*$. If the answer is B_{11}, and the move for $B_1 + B_{11}$ is B_2, it is natural that Myself asks what is the move for $A_2^* + B_2^*$, and so on.

But this is not what will happen if Myself tries to evaluate

$$\mathbf{CUT}(\mathbf{CUT}(A, A^* + B^*), B).$$

For this "protocol", Myself asks instead what is the move for $A_1^* + B_2^*$, forgetting completely what happened about A.

One can generally expect inefficiency (at least in the case of simple backtracking) if Myself follows the protocol

$$\mathbf{CUT}(\mathbf{CUT}(A, A^* + B^*), B).$$

In this case indeed, whenever the player associated with the formula B tries a new instantiation B_i for B, Myself comes back to the first instantiation for A, and asks what is the move for $A_1^* + B_i^*$.

Symmetrically, if Myself follow the protocol

$$\mathbf{CUT}(A, \mathbf{CUT}(A^* + B^*, B)).$$

In this case, whenever the player associated with the formula A tries a new instantiation A_i for A, Myself comes back to the first instantiation for B, and asks what is the move for $A_i^* + B_1^*$.

The symmetric protocol, which seems natural, is instead that Myself asks systematically the move for $A_i^* + B_j^*$ where A_i (resp. B_j) is the last instantiation of A (resp. B) that has been played. Furthermore, if Myself follows this protocol, then a given game can be analysed as two sequences of interaction for A and B that are interleaved, hence a direct termination argument.

Notice that this is a new "ternary" way of doing cut-elimination, which is not reducible to a combination of two "binary" cuts.

In the case where A or B is universal, then the "ternary" cut-elimination is equivalent to a combination of "two" cuts (except of doing less copies.) This is what happened, in a iterated way, with the cut-elimination procedure presented above compared to Gentzen's cut-elimination.

This analysis can be extended to the case of a "multi-cut" A, $A^* + B^*$, B, with A, B of arbitrary complexity, but the players follow a strategy of simple backtracking.

4 An inductive presentation of ω-logic

For sake of comparison, we reformulate usual definitions of ω-logic in the framework of generalised inductive definitions.

We define inductively when a multiset M of formulae is (classically) **true**. There are the following clauses:

- if M contains a true atomic formulae, then M is true,
- if there exists n_0 such that $M + A[n_0]$ is true, then $M + \exists n\ A[n]$ is true,
- if there exists n_0 such that $M + \exists n\ A[n] + A[n_0]$ is true, then $M + \exists n\ A[n]$ is true,
- if $M + A + B$ is true, then $M + A\ \vee\ B$ is true,
- if $M + A$ and $M + B$ are true, then $M + A\ \&\ B$ is true,
- if $M + A[n_0]$ is true for all integers n_0, then $M + \forall n\ A[n]$ is true.

Only the last clause is not finitary.

If we forget the point that in the game-theoretic presentation we consider only prenex formulae, the main difference is that in the game-theoretic presentation, we have to use the last rule whenever one formula is universal in the multiset of sequents. It follows that if there is a winning strategy for a configuration M, then M is true with the present definition.

Lemma 1 The following properties hold

- if A is a false atomic formula, and $M + A$ is true, then M is true,
- if $M + A\ \vee\ B$ is true, then $M + A + B$ is true,
- if $M + A\ \&\ B$ is true, then $M + A$ and $M + B$ are true,
- if $M + \forall n\ A[n]$ is true, then $M + A[n_0]$ is true for all integer n_0.

Proof: All these properties are of the form: if M is true, then M' is true, and they are proved directly by induction on the proof that M is true.

Lemma 2 If $M + A + A$ is true, then so is $M + A$.

Proof: By double induction: first on the formula A, and then by induction on the proof that $M + A$ is true, using the previous lemma 1.

Proposition 1 If $M + A$ and $N + A^*$ are true, then so is $M + N$.

Proof: By double induction, first on the formula A, and then by induction on the proofs that $M + A$ and $N + A^*$ are true. Let us look at one case: A is $\exists n\ B[n]$ and $M + A$ is true because $M + B[n_0] + A$ is true. Then, by lemma 1, we know that $N + B[n_0]^*$ is true. By induction hypothesis, by a cut between $N + A^*$ and $M + B[n_0] + A$, we get that $N + M + B[n_0]$ is true. By induction hypothesis, since $B[n_0]$ is less complex than A, we get that $N + N + M$ is true. By lemma 2, $N + M$ is true.

Conclusion

We have presented a conjecture of termination of an internal communication, that would refine Gentzen's cut-elimination. This conjecture is valid in the case of cuts of low logical complexity, and in a restricted case of "simple backtracking." The same idea in the case of multiple cuts leads to a protocol of cut-elimination distinct in general from the one where we decompose the multiple cut in binary cuts.

One important point to precise is the connection between the symmetric protocol we presented and the fact, noticed by Hugo Herbelin, that there does exist a Gentzen like cut-elimination procedure that lead to a symmetric answer. It is not clear at all yet what is the game-theoretical meaning of this procedure.

References

[1] Bernays P. (1968) "On the original Gentzen consistency proof for number theory." In Intuitionism and Proof Theory, Kino, Myhill, Vesley eds.

[2] Constable, R. "The semantics of evidence." Technical Report TR 85-684, Cornell University, Department of Computer Science, Ithaca, New York, May 1985.

[3] Gentzen G. (1969) "Collected Works." North-Holland, Szabo ed.

[4] Isles, D. "A Finite Analog to the Lowenheim-Skolem Theorem." Draft, 1990.

[5] Kleene, S.C. (1978) "Recursive Functionals and Quantifiers of Finite Types Revisited II." in The Kleene Symposium, North-Holland, J. Barwise, H.J. Keisler and K. Kunen eds.

[6] Martin-Löf P. (1968) "Notes on Constructive Mathematics." Almqvist & Wiksell ed.

[7] Martin-Löf P. "Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions." in Proceedings of the Second Scandinavian Logic Symposium, North-Holland, J.E. Fenstad ed.

[8] Ranta A., Ph. D. Thesis (1990), Stockholm.

[9] Tait W.W. (1968) "Normal Derivability in Classical Logic." Springer Lecture Notes in Mathematics 72, J. Barwise ed.

Logical Support for Modularisation†

Răzvan Diaconescu Joseph Goguen‡
Petros Stefaneas
Programming Research Group, Oxford University

July 1991

Abstract: Modularisation is important for managing the complex structures that can arise in large theorem proving problems, and in large software and/or hardware development projects. This paper studies some properties of logical systems that support the definition, combination, parameterisation and reuse of modules. These properties include: (1) the preservation of various kinds of conservative extension under pushouts; (2) distributive laws for information hiding over sums; and (3) (Craig style) interpolation properties. Our results illuminate some interesting differences between equational style logics and first order style logics, encouraging us to conclude that, on the whole, equational style logics are more suitable for modularisation than first order style logics. We represent logical systems as institutions, and modules as sets of sentences. An important technical foundation is a novel categorical axiomatisation of the notion of "inclusion". We also show how to subsume the deduction-based approach of π-institutions under that of ordinary institutions, so that our results also apply to that setting.

1 Introduction

Modules are an important feature of many modern programming and specification languages, assisting with the development of large and complex systems. Modularisation for theorem proving has been studied less, and in fact most results have appeared in the context of formal software engineering, particularly specification languages. This paper studies some properties of logical systems that are important as support for modularisation, where modules are represented as sets of sentences in a logical system, and logical systems are represented as institutions. This notion of module is directly applicable to theorem proving, software specification languages, hardware description languages, and logic-based programming languages; with some extra effort, it can even be applied to imperative programming languages. See [12] for some general discussion and examples.

We show that while a number of properties hold for any institution that satisfies certain modest conditions, some other properties are more delicate, and only hold for institutions that are close to equational logic. Moreover, certain properties depend crucially on how they are formulated; for example, forms of Craig interpolation that are equivalent for first order logic are not at all equivalent for many other important institutions. There are also some desirable properties that are rarely satisfied in practice, such as a strong form of distribution for information hiding over sum.

†The research reported in this paper has been supported in part by grants from the Science and Engineering Research Council and the Fujitsu Corporation.

‡Also, SRI International, Menlo Park CA 94025.

We have been particularly interested in the comparison between equational logic and classical first order logic. It has been argued that certain properties of equational logic render it unsuitable for use in specification. However, results and arguments in this paper suggest that these reasons are not really valid, and in fact, that equational logic may be superior in certain respects.

A categorical axiomatisation of the notion of inclusion permits us to give simple definitions for operations on modules in the general setting of institutions. Also, we show how producing a suitable model theory "out of thin air" for logical systems based on deduction (as formalised by π-institutions) allows them to be seen as ordinary institutions, so that our results about institutions also apply to logical systems that are based on deduction rather than on satisfaction.

1.1 Some Motivation

Experience shows that it can be very difficult to develop and maintain the logical structures that occur in large proofs and in large software and/or hardware development projects. One systematic approach to these difficulties is to *modularise* systems into relatively small, "mind sized" pieces that can be developed and maintained separately, and can then be interconnected to form the whole [11]. For many applications, it is sufficient to consider a module to be a set of sentences. This is the case in theorem proving, software and/or hardware specification, and in logic-based programming languages such as pure Lisp, pure Prolog, and OBJ. There are a number of important operations and relations on such modules. Two basic operations are *sum* and *renaming*. The first simply combines the resources available from two or more modules, while the second changes the names of some internal features. Laws concerning such operations could be used, for example, to prove the correctness of an efficient implementation of module interconnection for a programming language. A third operation which makes this challenging is *information hiding*, whereby some internal features of a module are made inaccessible outside the module. One would like to know that everything true of the visible part of some combination of modules can be proved from what is known about the visible parts of the component modules. Unfortunately, this often fails for real systems, as well as for theories over most institutions.

A particularly important relation between theories is that of *conservative extension*, which says that any model of the subtheory can be expanded to a model of the supertheory. This property can be important for the reuse of modules. Properties of extensions also arise in connection with parameterised (i.e., generic) modules.

A promising approach to system development (whether proofs, programs, or chips) is to start from some specification modules, and a description of the system as an interconnection of these modules. This permits the design, and the verification of certain properties, such as security, to be carried out at a more abstract level than that of code. It should thus improve reliability, and with suitable mechanical support, might also improve the efficiency of the programming process. In addition, it promotes software reuse, because some modules may be taken either directly from a library, or may be modifications of library modules. To support the construction of such a software development environment, it would be helpful to know what laws are satisfied by the basic operations on modules.

Institutions allow us to discuss the crucial relationship between theories and models, without a commitment to either side at the expense of the other. This is important, because there is a tension between these two aspects of logic, particularly for applications to Computing Science, because on the one hand, we are primarily interested in certain models, such as specific VLSI chips, or specific programs running on specific machines, but on the other hand, these may be very large, complex objects, of which only certain specific properties are of particular interest. To check such properties, we manipulate syntactic representations of the models and their properties. However, the criteria of correctness for these manipulations must refer back to the models.

It follows from this discussion that we should define a *module* to be a syntactic entity, such as a set of sentences, and that we should carefully distinguish such entities from their *denotations*, which will be sets of modules. It also follows that we should define notions like "conservative extension" of theories semantically, and then seek syntactic criteria for checking such relationships. Thus, we do not agree with the approach of Maibaum, Sadler, Turski and others [28, 46], who do not deal with models at all, but only with sentences and deduction. In our view, such an approach cannot adequately formalise the actual situation in (say) software engineering, which involves models of concrete objects, as well as their descriptions. This is particularly illustrated by the inability of the purely syntactic notion of conservative extension to capture the underlying semantic property. On the other hand, we also disagree with the approach of Sannella, Tarlecki and others [39, 40, 41] who define modules to be classes of modules, because such a definition does not capture the fact that real modules are bits of text, rather than (say) sets of running chips. This is because designers and verifiers necessarily deal with descriptions, rather than with the models themselves.

Under this view, operations on modules should be operations on sets of sentences. However, it is highly desirable that such operations should be meaningful as operations on the corresponding denotations, as opposed to merely being arbitary textual "edit" operations. Once again, semantics provides a "reality check" for syntax.

It is necessary to use machinery as abstract as categories and institutions for the comparative study of modules over various logical systems. For example, an assertion of the form "the Craig Interpolation Property and the Export Distributive Law are equivalent" is not meaningful in the context of a single logical system; it needs to be quantified over logical systems. Perhaps an analogy will help to clarify this point. Suppose that we are interested in studying properties of groups; two such properties might be "commutativity" and "finiteness." Then asserting that "commutativity and finiteness are equivalent" for some group where both properties happen to hold (such as Z_2) is valid, but not very meaningful: such an assertion should be universally quantified over groups (and then it is false).

1.2 Some History

The earliest work on software modularisation with which we are familiar is due to Parnas [34, 35, 33]. Modules differ from more traditional program structuring mechanisms such as subroutines, procedures and blocks, in that they may include a number of procedure and data definitions, may be parameterised, may import other modules, and may hide certain elements. Such modules can be separately compiled, and they support software reuse because they can also be separately specified and verified. A major motivation for modules in this sense is to facilitate the modification of software, by localizing the representation of data and the operations that depend upon it; this is called *information hiding*. This notion of module is essentially *syntactic*; it has to do with the texts that describe software systems. Thus, a semantic notion of module, such as that advocated by Sannella and Tarlecki [39, 40, 41], would fail to capture this basic aspect of modules.

The earliest work that we know on modules in specification is due to Burstall and Goguen, in connection with their specification language Clear [4, 5]. This approach has semantics based on the theory of institutions, and has been applied to various logic-based programming languages, particularly OBJ [8, 23] (which is based on order sorted equational logic), Eqlog [20] (which combines functional and logic programming), FOOPS [21] (which combines functional and object oriented programming), and FOOPlog [21] (which combines all three paradigms). In [11], this approach is extended to module systems for imperative programming languages.

The earliest work we know that gives a calculus of modules is due to Goguen and Burstall [14]. Building on previous work on Clear, they studied laws that relate *horizontal* structuring relationships, and *vertical* implementing (also called "refinement") relationships, concluding that the axioms of a 2-category should be satisfied. Clear introduced the ideas that a specification module is a theory, and that such modules are put together using colimits; these ideas have their origin in some earlier work on General Systems Theory [10, 17]. Clear provided operations for summing, renaming, extending, hiding, importing and (in the case of generics) applying modules. A number of general laws which apply in particular to module operations can be found in [10], and a number of others arise during the proofs in [5].

The so-called *module algebra* developed in the pioneering paper of Bergstra, Heering and Klint [2] represents another ambitious attempt to capture the horizontal structure of modules, by examining the equations that hold among certain basic operations upon modules, including sum and information hiding. These equations, together with constructors for signatures and sentences, form a many sorted equational presentation, about which some interesting results can be proved, including a normal form theorem. Unfortunately, this work has first order logic built into its choice of the constructors for signatures and sentences.

By contrast, the present paper starts with an arbitary logical system, and explores what properties it must have to support a practical calculus of modules. Every reasonable institution supports modules and certain simple operations on modules; we may then ask, what properties do these operations have? Note that this exploration should include properties that are not expressible as equational laws, such as the preservation of conservative extensions by pushouts.

Although [2] does not build on a general notion of logical system, it does abstract some interesting general principles from the special case of first order logic. In particular, we will see that the equivalence of Craig Interpolation with a distributive law for information hiding asserted in [2] is actually valid at the level of institutions, under certain conditions (namely, compactness and closure under implication and false); moreover, we can use essentially the same proof that they gave. Indeed, their paper was a major inspiration for the present paper, and in our opinion, opened a fascinating realm of new questions in the theory of institutions.

The original semantics of Clear [5] represented sums of theories by colimits, but failed to capture the use of subsignatures and subtheories that is natural in examples like equational logic and first order logic. The present paper shows how to capture such inclusions through a general axiomatisation of the notion of inclusion, such that sums are given by at least upper bounds of inclusions. A more concrete approach to inclusions is given by Sannella and Tarlecki in [39]. The desirability of using inclusions for theories first appeared in the thesis of Sannella [38], which gave a set theoretic semantics for Clear in the case of many sorted equational logic.

Another choice is whether to use *theories*, which are sets of sentences closed under entailment, or to use *presentations*, which are not necessarily so closed. Although the first choice represents the ideal, only finite sets can be actually implemented. However, designers and verifiers may need to know about consequences of a specification that do not happen to be in its original presentation, but instead appear in its theory. Therefore, a general approach should consider modules to be theories rather than presentations; then the problem for a practical implementation is how to manipulate finite sets of sentences in a way that respects the behaviours of their (generally infinite) corresponding closed theories, and of course, their models.

Much interesting work using institutions has been done by Tarlecki [42, 44, 45, 43] and by Sannella and Tarlecki [39, 40, 41]. See also [9] for some recent work on the semantics of module composition over institutions.

1.3 Summary

This paper includes the following:

1. a novel categorical characterisation of the notion of "inclusion," which is used to explicate the notion of "theory extension";
2. a lifting of exactness from the signatures of an institution to its theories;
3. an embedding of the category of π-institutions into the category of ordinary institutions;
4. a general discussion of Craig interpolation for institutions with inclusions;
5. a discussion of the intuitive significance of different distributive laws for information hiding over the sum of modules;
6. a proof that Craig interpolation is equivalent to a certain distributive law, for reasonable compact institutions with implication and false;
7. a discussion of the relationship between the syntactic and semantic formulations of conservative extension;
8. a proof that the middle distributive law holds for conservative extensions in reasonable semiexact institutions;
9. a proof that a pushout of a conservative (or persistent) extension is again conservative (or persistent) in a reasonable semiexact institution; and
10. a proof that the pushouts of two extensions conservative for initiality are again conservative for initiality in a reasonable semiexact institution.

1.4 Prerequisites

This paper assumes familiarity with basic category theory, including categories, functors, limits, colimits, and adjoints. The necessary material may be found in [27], [26], or in a more gentle style, [24]. By way of notation, we use ";" for composition, we let 1_A denote the identity morphism at an object A, and we let $|\mathbf{C}|$ denote the class of objects of a category $\mathbf{C}$.

We have found that certain readers may have doubts about set-theoretic foundations when category theory is used. In fact, this paper stays well away from anything that is genuinely problematical, and nearly any foundation that has been proposed for category theory will do, including the "hierarchy of universes" discussed e.g., by Mac Lane [27], Section I.6.

1.5 Acknowledgements

We wish to thank Drs. José Fiadeiro, Piet Rodenburg, and especially Andrzej Tarlecki, for their very useful comments on drafts of this paper. We also thank Prof. Rod Burstall for his important role in creating the theory of institutions. In addition, we thank Drs. José Meseguer, Donald Sannella, Andrzej Tarlecki, José Fiadeiro, Piet Rodenburg, and Jan Heering, as well as Profs. Thomas Maibaum, Jan Bergstra and Paul Klint for their inspiring papers, without which the present paper would not have been written.

2 Basic Concepts

This section presents some concepts that are needed for the rest of the paper. The first subsection discusses factorisations and inclusions, while the second summarises some aspects of institutions, particularly how they interact with inclusion systems. The third subsection discusses exactness, and the fourth π-institutions.

2.1 Factorisations and Inclusions

We first present the standard notion of a factorisation system (as in [26]), and then use it in building a novel axiomatisation for the notion of "inclusion" that is used in our work on modules.

Definition 1 An **image factorisation system** for a category C consists of a class $\mathcal{M}$ of monics and a class $\mathcal{E}$ of epics in C such that

(1) both $\mathcal{E}$ and $\mathcal{M}$ are closed under composition,

(2) all isomorphisms are in both $\mathcal{M}$ and $\mathcal{E}$, and

(3) every morphism f in C can be factored as $e;m$ with $e \in \mathcal{E}$ and $m \in \mathcal{M}$ "uniquely up to isomorphism" in the sense that if $e';m'$ is another factorisation of f with $e' \in \mathcal{E}$ and $m' \in \mathcal{M}$, then there is a unique isomorphism from the **centre object** C of $e;m$ (i.e., the target of e and the source of m) to the centre object C' of $e';m'$ such that the diagram below commutes:

$$\begin{tikzcd} & C \arrow[dr, "m"] \arrow[dd] & \\ A \arrow[ur, "e"] \arrow[dr, "e'"'] & & B \\ & C' \arrow[ur, "m'"'] & \end{tikzcd}$$

□

Our definition of inclusion system will use the following, which comes from [1]:

Definition 2 Let $\mathcal{M}$ be a class of arrows in a category C. Then C is $\mathcal{M}$-**inductive** iff whenever all the injections of a co-cone over some diagram in C lie in $\mathcal{M}$, then there is a colimit of this diagram such that the universal arrow from the colimit to the co-cone also lies in $\mathcal{M}$. □

Now the main concept of this subsection:

Definition 3 An **inclusion system** for a category C consists of a class I of monics and a class $\mathcal{E}$ of epics in C such that:

(1) both $\mathcal{E}$ and I are subcategories of C such that $|\mathcal{E}| = |\mathrm{I}| = |\mathrm{C}|$;

(2) all isomorphisms in I are identities, and all isomorphisms in C are in $\mathcal{E}$;

(3) every morphism f in C can be factored uniquely as $e;i$ with $e \in \mathcal{E}$ and $i \in \mathrm{I}$;

(4) I is a partial order (i.e., for any objects A, B, there is at most one morphism $A \to B$ in I and if there is also a morphism $B \to A$ in I, then $A = B$);

(5) I has finite least upper bounds (i.e., finite coproducts in I), denoted $+$; and

(6) C is I-inductive.

We will call the morphisms in I **inclusions**, and use the notation $A \hookrightarrow B$. Also, we let $\emptyset$ denote the bottom element of I (i.e., the empty least upper bound, which is also the initial object in I). Given $i\colon A \hookrightarrow B$ and $f\colon B \to C$, we let $f^{-}{}_{A}$ denote the composition $i;f$, and call it the **restriction** of f to A. We will also say that a category with an inclusion system is **inclusive**. □

Most of the categories used in specification and programming have obvious and natural inclusion systems, including the following:

Example 4 The category *Set* of sets with I the inclusions and $\mathcal{E}$ the surjections is an inclusion system. □

Example 5 Recall (e.g., from [22]) that a many sorted signature Σ consists of a set S of sorts and an $S^* \times S$-indexed family of sets $\Sigma_{w,s}$ whose elements are the operation symbols of arity (i.e., input sorts) $w \in S^*$, and value (i.e., output) $s \in S$. Also, a morphism $(S, \Sigma) \to (S', \Sigma')$ of many sorted signatures consists of a function $f\colon S \to S'$ and a $S^* \times S$-indexed family of functions $g_{w,s}\colon \Sigma_{w,s} \to \Sigma'_{f(w),f(s)}$. Then we get an inclusion system in a natural way by defining I by $(S, \Sigma) \hookrightarrow (S', \Sigma')$ iff $S \hookrightarrow S'$ and $\Sigma_{w,s} \hookrightarrow \Sigma'_{w,s}$ for each $w \in S^*$ and $s \in S$, and defining $\mathcal{E}$ to be the (component-wise) surjective signature morphisms. □

Example 6 Similarly, the usual category of (unsorted) first order signatures has an obvious inclusion system. Recall that a first order signature is a pair (Σ, Π), where both Σ are Π are ω-indexed families of sets, with Σ_n the operation symbols of arity n and Π_n the predicate symbols of arity n, for $n \in \omega$. Also, a morphism $(\Sigma, \Pi) \to (\Sigma', \Pi')$ is a pair (f, g) of ω-indexed functions, where $f_n\colon \Sigma_n \to \Sigma'_n$ translates operation symbols, and $g_n\colon \Pi_n \to \Pi'_n$ translates predicate symbols. Then we get an inclusion system by defining I by $(\Sigma, \Pi) \hookrightarrow (\Sigma', \Pi')$ iff $\Sigma_n \hookrightarrow \Sigma'_n$ and $\Pi_n \hookrightarrow \Pi'_n$ for each $n \in \omega$, and defining $\mathcal{E}$ to be the (component-wise) surjective signature morphisms. □

The inclusion systems axiom are sufficient to ensure that many familiar properties of inclusions hold in general, including the following:

Fact 7 I is I-**right cancellable**, in the sense that if $f;g \in \mathrm{I}$ and $g \in \mathrm{I}$, then $f \in \mathrm{I}$.

Proof: Factor f as $e;i$ with $e \in \mathcal{E}$ and $i \in \mathrm{I}$. Then $e;i;g$ is a factorisation of $f;g$, with $i;g \in \mathrm{I}$ because both i and g are inclusions. But $f;g$ also factors trivially as $1;(f;g)$, and so (3) of Definition 3 gives $e = 1$ and $f = i$. Therefore, $f \in \mathrm{I}$. □

Fact 8 If C is a category with an inclusion system $(\mathrm{I}, \mathcal{E})$, then its category I of inclusions has all finite colimits. If C has an initial object $\emptyset$, then $\emptyset$ is also initial in I; hence, C has at most one initial object.

Proof: I trivially has coequalisers because it is a partial order, and it has finite coproducts by (5) of Definition 3. $\square$

Next, we discuss how the greatest lower and least upper bounds of objects in I relate to constructions in C; these arguments have a set-theoretic flavour.

Proposition 9 Let C be a category with an inclusion system $(\mathrm{I}, \mathcal{E})$ and with pullbacks. Let $A \hookrightarrow B$ and $A' \hookrightarrow B$ be inclusions in C. Then there is a unique pullback in C of these inclusions such that the arrows of the pullback cone are also inclusions.

Proof: Let $(P \to A, P \to A')$ be a pullback in C of the inclusions of A and A' into B. Now factor the pullback arrows as $P \to A = e; C \hookrightarrow A$ and $P \to A' = e'; C' \hookrightarrow A'$ where $e, e' \in \mathcal{E}$. Because of the uniqueness of the factorisation of $P \to B$, it follows that $C = C'$ and $e = e'$. It now follows that $(C \hookrightarrow A, C \hookrightarrow A')$ is also a pullback cone, and the uniqueness of C follows from Fact 7. $\square$

Definition 10 Let C be a category with an inclusion system $(\mathrm{I}, \mathcal{E})$ and with pullbacks. Let A and B be objects of C. Then the **sum** of A and B, denoted $A + B$, is the coproduct of A and B in I, and the **intersection** of A and B, denoted $A \cap B$, is the (unique) inclusion pullback in C of the inclusions $(A \hookrightarrow A + B, B \hookrightarrow A + B)$ given by Proposition 9. $\square$

Proposition 11 Let C be a category with an inclusion system $(\mathrm{I}, \mathcal{E})$ and with pullbacks. Then for any objects $A, B \in |\mathrm{C}|$ we have:

1. the sum $A + B$ is the least upper bound of A and B in I;
2. the intersection $A \cap B$ is the greatest lower bound of A and B in I;
3. sum and intersection are unique;
4. $A \cap B$ is included in A and B, and all three of these objects are included in $A + B$;
5. the intersection is the pullback in C of the sum; and
6. if C has pushouts, then the sum is the pushout in C of the intersection.

Proof: The first two assertions follow directly from (4) of Definition 3. Then the sum $A + B$ is unique by its definition as the least upper bound in I of A and B, and the intersection is unique because it is the greatest lower bound in I.

The maps $A \cap B \hookrightarrow A$ and $A \cap B \hookrightarrow B$ are inclusions by Propositions 9, and thus $A \cap B \hookrightarrow A + B$ is a composite of inclusions.

That the intersection is the pullback in C of the sum follows from the definition of intersections and the previous proposition.

Now let Q be a pushout of the intersection. Let $u \colon Q \to A + B$ be the unique arrow such that $(A \to Q); u = (A \hookrightarrow A + B)$ and $(B \to Q); u = (B \hookrightarrow A + B)$. Because C is I-inductive, we can choose Q such that $Q \to A + B$ is also an inclusion. It then follows that both $A \to Q$ and $B \to Q$ are inclusions by right cancellability. Because $A + B$ is the least upper bound of A and B, and because $A, B \subseteq Q \subseteq A + B$, we conclude that $Q = A + B$. $\square$

Definition 12 An inclusion system I for a category $\mathcal{C}$ is **distributive** iff the following two laws hold for all objects A, B, C in $\mathcal{C}$:

1. $A + (B \cap C) = (A \cap B) + (A \cap C)$.
2. $A \cap (B + C) = (A + B) \cap (A + C)$.

□

We can get a non-distributive inclusion system from any non-distributive lattice L, by letting $\mathcal{C}$ be L regarded as the partial order category with an arrow $A \to B$ iff $A \leq B$ in L.

Definition 13 Let C and C′ be categories with inclusion systems $(\mathrm{I}, \mathcal{E})$ and $(\mathrm{I}', \mathcal{E}')$, respectively. Then an **inclusion system morphism** is a functor $F\colon \mathrm{C} \to \mathrm{C}'$ such that $F^{-}{}_{\mathrm{I}}\colon \mathrm{I} \to \mathrm{I}'$, and $F^{-}{}_{\mathcal{E}}\colon \mathcal{E} \to \mathcal{E}'$, and $\cap$ and $+$ are preserved. □

Fact 14 If F is a morphism of inclusion systems, then $F(c) = F(i); F(e)$ is the $(\mathrm{I}', \mathcal{E}')$-factorisation of $F(c)$ in C′ if $i; e$ is the $(\mathrm{I}, \mathcal{E})$-factorisation of c in C. □

The following expresses a basic relationship between pushouts and inclusions that is mainly used in Section 5:

Definition 15 Let C be an inclusive category. Then **pushouts preserve inclusions** in C iff whenever a pair of arrows $(A \hookrightarrow A', A \to B)$ has a pushout, then they have a pushout of the form $(B \hookrightarrow B', A' \to B')$. □

2.2 Institutions

In a series of papers beginning in 1979, Burstall and Goguen have developed institutions as a formalisation of the intuitive notion of a logical system; the most recent and complete exposition is [16]. This approach gives axioms for the satisfaction relation $\models_\Sigma$ that take account of the fact that one may wish to translate sentences and/or models from one signature to another. A special case of this translation may be familiar from first order model theory: let $\Sigma \hookrightarrow \Sigma'$ be an inclusion of first order signatures, and let M be a Σ'-model; then we can form $M^{-}{}_{\Sigma}$, the *reduct* of M to Σ. Similarly, if e is a Σ-sentence, then we can always view it as a Σ'-sentence (but there is no standard notation for this). The key axiom, called the Satisfaction Condition, says that truth is invariant under change of notation. This paper uses the precision of this formalisation of logical systems to discuss properties of modules over logical systems in a general way.

Definition 16 An **institution** consists of

1. a category $Sign$, whose objects are called **signatures**,
2. a functor $Sen\colon Sign \to Set$, giving for each signature a set whose elements are called **sentences** over that signature,
3. a functor $Mod\colon Sign \to Cat^{op}$ giving for each signature Σ a category whose objects are called Σ-**models**, and whose arrows are called Σ-(**model**) **morphisms**, and
4. a relation $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$ for each $\Sigma \in |Sign|$, called Σ-**satisfaction**,

such that for each morphism $\varphi\colon \Sigma \to \Sigma'$ in $Sign$, the **Satisfaction Condition**

$$M' \models_{\Sigma'} Sen(\varphi)(e) \quad \text{iff} \quad Mod(\varphi)(M') \models_{\Sigma} e$$

holds for each $M' \in |Mod(\Sigma')|$ and $e \in Sen(\Sigma)$.

An institution is **compact** iff whenever $E \models e$ then there is a finite subset $E' \subseteq E$ such that $E' \models e$.

We may write $\varphi(e)$ or even φe for $Sen(\varphi)(e)$, and $M'\!\upharpoonright_{\varphi}$ or possibly $\varphi(M')$, for $Mod(\varphi)(M')$; also we may drop the signature subscripts on the satisfaction relation when it is not confusing. If M is a Σ-model and E is a set of Σ-sentences, then we let $M \models_{\Sigma} E$ mean that $M \models_{\Sigma} e$ for each $e \in E$. Also, if E is a set of Σ-sentences and e is a single Σ-sentence, then we let $E \models_{\Sigma} e$ mean that for each Σ-model M, if $M \models_{\Sigma} E$ then $M \models_{\Sigma} e$. Finally, we let $\varphi(E) = \{\varphi(e) \mid e \in E\}$. $\Box$

Definition 17 An institution with an inclusion system $(\mathrm{I},\mathcal{E})$ on its category of signatures **respects** that inclusion system iff the functor $Sen\colon Sign \to Set$ is an inclusion system morphism, where Set has the inclusion system described in Example 4; for short, let us say that such an institution is **inclusive**. Given $i\colon \Sigma \hookrightarrow \Sigma'$ and a Σ'-model M, we may write $i(M)$ as $M\!\upharpoonright_{i}$ or $M\!\upharpoonright_{\Sigma}$ and call it the **reduct** of M to Σ.

An inclusive institution is **distributive** iff the inclusion system on its signatures is distributive. $\Box$

Example 18 The many sorted algebraic institution is inclusive and distributive with the inclusion system of Example 5 on its signatures. $\Box$

Example 19 The unsorted first order institution is inclusive and distributive with the inclusion system of Example 6 on its signatures. $\Box$

Any non-distributive inclusion system can be turned into a trivial non-distributive inclusive institution, with the inclusion system as its category of signatures, and with no models and no sentences. However, all of the usual logical systems are distributive inclusive institutions, including first-order logic and equational logic.

Definition 20 A **presentation** is a pair (Σ, E) where Σ is a signature and E is a set of Σ-sentences. A presentation (Σ, E) is a **theory** iff E is **closed** under semantic entailment, in the sense that if $E \models_{\Sigma} e$ then $e \in E$. We let $E^{\bullet}$ denote the closure of a set E of sentences[1].

If $T = (\Sigma, E)$ is a theory, then its **denotation**, which we might write $[\![T]\!]$, is the full subcategory of $Mod(\Sigma)$ with objects M such that $M \models_{\Sigma} E$.

If (Σ, E) and (Σ', E') are presentations, then a **presentation morphism** $(\Sigma, E) \to (\Sigma', E')$ is a signature morphism $\varphi\colon \Sigma \to \Sigma'$ such that $e \in E$ implies $\varphi(e) \in E'^{\bullet}$. Let $Pres(\mathcal{I})$ denote the category of presentations over an institution $\mathcal{I}$, and let $Th(\mathcal{I})$ denote its subcategory of theories over $\mathcal{I}$.

A presentation morphism $\varphi\colon (\Sigma, E) \to (\Sigma', E')$ over an inclusive institution is an **inclusion** iff φ is an inclusion; note that in this case, $E \subseteq E'$. We may also call such morphisms **extensions**. $\Box$

The following basic result is proved in [16], and the one below it has a similar flavour:

[1] It is easy to see that each presentation is contained in a least closed set of sentences.

Theorem 21 If the category of signatures of an institution $\mathcal{I}$ has [finite] colimits, then so does its category $Th(\mathcal{I})$ of theories. □

Proposition 22 If $\mathcal{I}$ is an institution with a factorisation system $(\mathrm{I}, \mathcal{E})$ on its signatures, then $(\mathrm{I}_{Th}, \mathcal{E}_{Th}) = (\{\Phi \in Th \mid \Phi \in \mathrm{I}\}, \{\Phi\colon (\Sigma, E) \to (\Sigma', E') \in Th \mid \Phi \in \mathcal{E} \text{ and } (\Phi(E))^\bullet = E'\})$ is a factorisation system on its theories. If $\mathcal{I}$ respects an inclusion system $(\mathrm{I}, \mathcal{E})$ on its signatures, and if its category of signatures has finite colimits, then $(\mathrm{I}_{Th}, \mathcal{E}_{Th})$ is an inclusion system for $Th(\mathcal{I})$.

Proof: Given a theory morphism $\Phi\colon (\Sigma, E) \to (\Sigma', E')$, let $\Phi = e; m$ be its factorisation in $Sign$, with centre object Σ''. Let E'' be the closure of $e(E)$. Then $e\colon (\Sigma, E) \to (\Sigma'', E'')$ and $m\colon (\Sigma'', E'') \to (\Sigma', E')$ are theory morphisms, and so $e; m$ is a factorisation of Φ in Th. It is easy to check that this forms a factorisation system.

If $\mathcal{I}$ respects the inclusion system $(\mathrm{I}, \mathcal{E})$, then (2), (3) and (4) of Definition 3 hold trivially, and I_{Th} has finite coproducts by an argument similar to that given for Theorem 21 in [16]. The same approach works for condition (6). □

Fact 23 Under the hypothesis of the Proposition 22, the forgetful functor $Th(\mathcal{I}) \to Sign$ is an inclusion system morphism. Moreover, it is limit and colimit preserving. □

The assumptions encapsulated in the following play a basic role in this paper:

Definition 24 An institution is **reasonable** iff it is inclusive and its signature category has finite colimits and finite limits. □

All of the institutions that are widely used in Computing Science are reasonable, including classical first order logic and equational logic, in both sorted and unsorted variants.

Corollary 25 If (Σ, E) and (Σ', E') are theories over a reasonable institution, then

1. $(\Sigma, E) + (\Sigma', E') = (\Sigma + \Sigma', (E \cup E')^\bullet)$;
2. $(\Sigma, E) \cap (\Sigma', E') = (\Sigma \cap \Sigma', Sen(\Sigma) \cap Sen(\Sigma') \cap E \cap E')$.

Proof: The first equation follows from the proof of Theorem 21, and the same technique works for intersection, except that we do not need to take the closure because the intersection of the $(\Sigma \cap \Sigma')$-theories $Sen(\Sigma) \cap E'$ and $Sen(\Sigma') \cap E$ is already a $(\Sigma \cap \Sigma')$-theory. □

Following [16], we now present institution morphisms, which are useful for comparing and for transferring results among different logical systems.

Definition 26 Let $\mathcal{I}$ and $\mathcal{I}'$ be institutions. Then an **institution morphism** $\Phi\colon \mathcal{I} \to \mathcal{I}'$ consists of

1. a functor $\Phi\colon Sign \to Sign'$,
2. a natural transformation $\alpha\colon \Phi; Sen' \Rightarrow Sen$, and
3. a natural transformation $\beta\colon Mod \Rightarrow \Phi^{op}; Mod'$

such that the following **Satisfaction Condition** holds

$$M \models_\Sigma \alpha_\Sigma(e') \text{ iff } \beta_\Sigma(M) \models'_{\Phi(\Sigma)} e'$$

for any Σ-model M from $\mathcal{I}$ and any $\Phi(\Sigma)$-sentence e' from $\mathcal{I}'$. Institutions with institution morphisms form a category, which we denote *INS*. □

2.3 Exactness

An important model theoretic property of many logical systems is that finite colimits are preserved by the model functor. Thus, if we combine some theories T_i in a diagram $T\colon I \to Th(\mathcal{I})$ having colimit (i.e., result of combination) C, then the denotations of the T_i and C behave in the way one would hope: $Mod(C)$ is the limit of the diagram $Mod^{op};T\colon I \to Cat$. In particular (and assuming that the categories of Σ-models are concrete), our intuition would lead us to hope that a model of $T_1 \bigoplus T_2$ (the co-product) would consist of a pair of models, one of T_1 and the other of T_2; i.e., we intuitively expect $Mod(T_1 \bigoplus T_2)$ to be $Mod(T_1) \times Mod(T_2)$. The situation is similar for a pushout of theory morphisms $T_0 \to T_1$ and $T_0 \to T_1$, which for simplicity we assume are theory inclusions, so that T_0 is shared between T_1 and T_2: we expect that a model of $T_1 \bigoplus_{T_0} T_2$ (the pushout) can be constructed from a pair of models, one of T_1 and the other of T_2, by identifying their reducts to T_0; that is, we expect $Mod(T_1 \bigoplus_{T_0} T_2)$ to be the pullback of $Mod(T_1) \to Mod(T_0)$ and $Mod(T_2) \to Mod(T_0)$. This property, which we call *exactness*, seems to have first arisen in [41], and is also used in the pioneering work of Tarlecki [44] on abstract algebraic institutions, and of Meseguer [31] on categorical logics[2].

Definition 27 An institution is **exact** iff the model functor $Mod\colon Sign \to Cat^{op}$ preserves finite colimits, and is **semiexact** iff Mod preserves pushouts. □

Although many sorted logics tend to be exact, their unsorted variants tend to be semiexact. In particular, the model functor does not preserve coproducts for either unsorted first order logic or unsorted equational logic. This is undesirable from the point of view of modularisation, and is related to the well known fact that the sum of terminating term rewriting systems need not be terminating. A general conclusion is that unsorted logics are unnatural for many applications in Computing Science.

We have recently proved that any chartered institution is exact. Charters were introduced by Goguen and Burstall [15] as a general way to produce institutions. The basic intuition is that the syntax of a logical system is an initial algebra. Because it appears that most institutions of interest in Computing Science can be chartered, it follows that most institutions of interest in Computing are exact. In particular, both many sorted first order logic and many sorted equational logic are exact.

Notice that, for any institution $\mathcal{I}$, the model functor Mod extends to $Th(\mathcal{I})$, by mapping a theory (Σ, E) to the full subcategory $Mod(\Sigma, E)$ of $Mod(\Sigma)$ formed by the Σ-models that satisfy E. The following result shows that one can lift exactness from signatures to theories, so that exactness depends only on the behaviour of signatures, and is independent of what happens with sentences. Semiexactness for theories plays an important role in the "categorical logics" described by Meseguer in [31]. Here, we show that this follows from the corresponding property for signatures:

Proposition 28 If an institution is semiexact, then $Mod\colon Th \to Cat^{op}$ preserves pushouts.

Proof: Let $\varphi_1 : (\Sigma', E') \to (\Sigma_1, E_1)$ and $\varphi_2 : (\Sigma', E') \to (\Sigma_2, E_2)$ be morphisms of theories and let $\varphi_1' : (\Sigma_2, E_2) \to (\Sigma, E)$ and $\varphi_2' : (\Sigma_1, E_1) \to (\Sigma, E)$ be their pushout. Recall [16] that (φ_1', φ_2') is the pushout of (φ_1, φ_2) in $Sign$ and E is the closure of $\varphi_2'(E_1) \cup \varphi_1'(E_2)$.

Let M_1 be a Σ_1-model of E_1 and M_2 a Σ_2-model of E_2 such that $M_1{}^{-}{}_{\varphi_1} = M_2{}^{-}{}_{\varphi_2}$; now let M' denote this Σ'-model. Then by the Satisfaction Condition, M' satisfies E'. By semiexactness

[2] Meseguer [31] introduced the term exactness, but used it for the concept that we call semiexactness here.

and the construction of pullbacks in Cat, there is a Σ-model M such that $M^{-}{}_{\varphi_2'} = M_1$ and $M^{-}{}_{\varphi_1'} = M_2$. By the Satisfaction Condition again, M satisfies the translations of both E_1 and E_2, and thus satisfies E. We have now shown that any pair of models (M_1, M_2) with $M_1 \in |Mod(\Sigma_1, E_1)|$ and $M_2 \in |Mod(\Sigma_2, E_2)|$ and $M_1{}^{-}{}_{\varphi_1} = M_2{}^{-}{}_{\varphi_2}$ determines a (Σ, E)-model M.

Conversely, any (Σ, E)-model M is determined in this way by its translations $M_1 = M^{-}{}_{\varphi_2'}$ and $M_2 = M^{-}{}_{\varphi_1'}$ which, by the Satisfaction Condition, satisfy E_1 and E_2, respectively.

Because the models of a theory form a full subcategory of the models of its signature, we can extend this argument to model morphisms. By semiexactness, ${}^{-}{}_{\varphi_2'} \colon Mod(\Sigma, E) \to Mod(\Sigma_1, E_1)$ and ${}^{-}{}_{\varphi_1'} \colon Mod(\Sigma, E) \to Mod(\Sigma_2, E_2)$ are the pullback of ${}^{-}{}_{\varphi_1} \colon Mod(\Sigma_1, E_1) \to Mod(\Sigma', E')$ and ${}^{-}{}_{\varphi_2} \colon Mod(\Sigma_2, E_2) \to Mod(\Sigma', E')$. $\Box$

Corollary 29 If an institution is exact, then $Mod \colon Th \to Cat^{op}$ preserves finite colimits.

Proof: By exactness, Mod maps the initial object of $Sign$ to the terminal (singleton) category. Because the only model of this category satisfies the empty theory (i.e., the tautologies over the initial signature) we conclude that the model functor maps the initial theory to the terminal category. Now we are done, because all finite colimits can be constructed from pushouts and an initial object. $\Box$

The following result will later help us with pushouts of conservative extensions.

Fact 30 In an inclusive institution, if pushouts preserve inclusions of signatures, then pushouts also preserve inclusions of theories.

Proof: This follows from the fact that the forgetful functor $Th(\mathcal{I}) \to Sign$ creates both colimits [16] and inclusions. $\Box$

2.4 π-Institutions

It has been claimed (e.g., in [28]) that the standard notion of institution (Definition 16) is not suitable for the foundations of software engineering, if one takes an approach based purely on deduction, rather than on satisfaction. Following the work of Fiadeiro and Sernadas [7], such logical systems can be formalised as π-institutions, which have for each signature Σ a set of Σ-sentences, but no models. To compensate for the lack of models, a consequence relation is given on sentences. We will use the definition of Fiadeiro and Sernadas [7] as modified by Meseguer [31], rather than that of Maibaum and Fiadeiro [29], which seems overly complex to us. Also note that Harper, Sannella and Tarlecki [25] have given a definition that is similar to Meseguer's, but restricted to finite sets of sentences.

Meseguer [31] showed how to construct an institution from a system of deduction, by producing a model theory "out of thin air" using a comma category construction on theories. We extend this construction to a functor, and conclude that π-institutions are just a special case of ordinary institutions, so that the results of this paper apply directly to the deduction-based approach to formal software engineering advocated by Turski and Maibaum [46] and Maibaum and Sadler [28], and later formalized by Fiadeiro and Maibaum using π-institutions [29].

Definition 31 A **π-institution** consists of

1. a category $Sign$, whose objects are called **signatures**,

2. a functor $Sen\colon\ Sign \to Set$, giving for each signature a set whose elements are called **sentences** over that signature, and

3. a relation[3] $\vdash_\Sigma \subseteq \mathcal{P}(Sen(\Sigma)) \times Sen(\Sigma)$ for each $\Sigma \in |Sign|$, called Σ-**consequence**,

such that the following conditions hold:

A. **reflexivity:** $\{e\} \vdash_\Sigma e$ for each $e \in Sen(\Sigma)$;

B. **monotonicity:** if $E \vdash_\Sigma e$ and $E \subseteq E'$ then $E' \vdash_\Sigma e$;

C. **transitivity:** if $E \vdash_\Sigma e'$ for each $e' \in E'$ and if $(E \cup E') \vdash_\Sigma e$, then $E \vdash_\Sigma e$;

D. **translation:** if $E \vdash_\Sigma e$ and if $\varphi\colon\ \Sigma \to \Sigma'$ in $Sign$, then $\varphi(E) \vdash_{\Sigma'} \varphi(e)$.

A π-institution is **compact** iff whenever $E \vdash_\Sigma e$ then there is some finite $E' \subseteq E$ such that $E' \vdash_\Sigma e$. □

Notice that although π-institutions axiomatise a syntactic consequence relation, they do not axiomatise an underlying notion of deduction.

Definition 32 Let $\mathcal{I}$ and $\mathcal{I}'$ be π-institutions. Then a π-**institution morphism** $\Phi\colon\ \mathcal{I} \to \mathcal{I}'$ consists of

1. a functor $\Phi\colon\ Sign \to Sign'$, and

2. a natural transformation $\alpha\colon\ \Phi; Sen' \Rightarrow Sen$,

such that $E \vdash_{\Phi(\Sigma)} e$ implies $\alpha_\Sigma(E) \vdash_\Sigma \alpha_\Sigma(e)$ for any signature Σ, for any set E of $\Phi(\Sigma)$-sentences, and any $\Phi(\Sigma)$-sentence e. Let π-*INS* denote the category of π-institutions with π-institution morphisms. □

There is a forgetful functor $\mathcal{U}\colon\ INS \to \pi\text{-}INS$ which maps any institution to a corresponding deductive system, by forgetting the models, and by letting $E \vdash e$ mean $E \models e$. Applying $\mathcal{U}$ to an institution morphism forgets its model part, and the Satisfaction Condition for institution morphisms helps to prove that this indeed gives a π-institution morphism.

In [31], Meseguer gives a comma category construction for categories of models of a π-institution. It is remarkable that this construction gives exactly the right models for the usual examples. The following result extends this construction to institution morphisms, thus giving a functor that embeds the category of π-institutions as a retract into the category of institutions. This enables us to apply our results about institutions to π-institutions.

Theorem 33 The forgetful functor $\mathcal{U}\colon\ INS \to \pi\text{-}INS$ has a retract (i.e., a left inverse functor) $\mathcal{F}\colon\ \pi\text{-}INS \to INS$.

Proof: The functor $\mathcal{F}$ maps a π-institution $(Sign, Sen, \vdash)$ to the institution $(Sign, Sen, Mod, \models)$ with the same category of signatures and the same sentence functor. Given a signature Σ, the category $Mod(\Sigma)$ is defined to be the comma category $(\Sigma, ?^\bullet)/Th$. Thus, a Σ-model is just a theory morphism $G_1 : (\Sigma, ?^\bullet) \to (\Sigma_1, E_1)$, and a model morphism $H : G_1 \to G_2$ is just a theory morphism $(\Sigma_1, E_1) \to (\Sigma_2, E_2)$ such that $G_2 = G_1; H$.

[3]Here $\mathcal{P}$ denotes the power set function.

If $D\colon \Sigma' \to \Sigma$ is a signature morphism, then a Σ-model G is translated to the Σ'-model $D;G$. The satisfaction of a Σ-sentence e by any Σ-model $G_1\colon (\Sigma, ?^\bullet) \to (\Sigma_1, E_1)$ is defined by $G_1(e) \in E_1$. It is easy to check that these data define an institution.

Now consider a π-institution morphism $(\Phi, \alpha)\colon (Sign, Sen, \vdash) \to (Sign', Sen', \vdash')$. We let $\mathcal{F}$ map this to $(\Phi, \alpha, \beta)\colon (Sign, Sen, Mod, \models) \to (Sign', Sen', Mod', \models')$, where only the natural transformation $\beta\colon Mod \Rightarrow \Phi^{op}; Mod'$ has still to be defined. For any signature Σ, β_Σ maps the model $G_1\colon (\Sigma, ?^\bullet) \to (\Sigma_1, E_1)$ to $\Phi(G_1)\colon (\Phi(\Sigma), ?^\bullet) \to (\Phi(\Sigma_1), \alpha^{-1}_{\Sigma_1}(E_1))$. That $\alpha^{-1}_{\Sigma_1}(E_1)$ is a theory follows from the Satisfaction Condition on α. Also, a model morphism $H\colon G_1 \to G_2$ is mapped to the model morphism $\Phi(H)\colon \Phi(G_1) \to \Phi(G_2)$, and $\Phi(H)$ is indeed a morphism of theories $(\Phi(\Sigma_1), \alpha^{-1}_{\Sigma_1}(E_1)) \to (\Phi(\Sigma_2), \alpha^{-1}_{\Sigma_2}(E_2))$, by the naturality of α.

To show the Satisfaction Condition for institution morphisms, we have to check that for any Σ-model $G_1\colon (\Sigma, ?^\bullet) \to (\Sigma_1, E_1)$ and any $\Phi(\Sigma)$-sentence e', we have $G_1 \models_\Sigma \alpha_\Sigma(e')$ iff $\beta_\Sigma(G_1) \models'_{\Phi(\Sigma)} e'$. This condition is equivalent to $G_1(\alpha_\Sigma e') \in E_1$ iff $\Phi(G_1)e' \in \alpha^{-1}_{\Sigma_1}(E_1)$. This holds because $\alpha_\Sigma; G_1 = \Phi(G_1); \alpha_{\Sigma_1}$ by the naturality of α.

This completes the definition of $\mathcal{F}$, and it remains only to check that $\mathcal{F};\mathcal{U}$ is the identity. $\square$

3 Module Algebra

This section and the next discuss some equations that may be satisfied by modules (i.e., theories) over a reasonable institution $\mathcal{I}$. Recall we have shown that $Th(\mathcal{I})$ also has an inclusion system and finite colimits.

In large software and/or hardware development projects, it is important to avoid accidental clashes between the names of operations. One way to do this is to consider that the "real name" of an operation is qualified by the name of the module where it is defined; then one can use short names when there are no clashes, and one can avoid clashes by using the long names. This solution has been implemented in the OBJ3 system [23] simply by renaming all operations as they are stored into the database. Then module importation is just inclusion of the renamed theories. Thus, our mathematical treatment of module importation by theory inclusion does not have to be modified to take account of the practical difficulties of real world projects.

We will use the following notations: if $T = (\Sigma, E)$, then we let $\Sigma(T) = \Sigma$; and given a signature Σ, we let $T(\Sigma) = (\Sigma, \emptyset^\bullet)$ and we call it the **empty theory**. Notice that even for equational logic, there may be some sentences in $T(\Sigma)$, for example, those of the form $(\forall x)\ x = x$.

3.1 Sum

A very basic operation on theories is simply to *combine* their features into a single theory. This was modelled with colimits in Clear [5] in a way that permitted shared subtheories. For an inclusive institution, we can use the specific colimit that is given by the sum.

Recall that given theories T and T', their sum is their coproduct $T + T'$ in I_{Th}. In examples from concrete institutions such as many sorted equational logic and first order logic, $T + T'$ is in general not a disjoint union, because there may be some sharing between T and T'.

Proposition 34 Given theories T, T', T'', then:

1. $T + T' = T' + T$.

2. $T + (T' + T'') = (T + T') + T''$.

3. $T + \emptyset = T$.

4. $T + T = T$.

Proof: These are immediate from Proposition 22. $\Box$

3.2 Renaming

In the practical development of large proofs, or large hardware and/or software systems, it can be very helpful to *reuse* theories and code. However, sometimes the source texts will not have the right names for the intended reuse; for example, we may wish to avoid having some variables be shared between two given modules. In such circumstances, it is important to be able to *rename* the features of modules. The operation $\star$ described below accomplishes this. (Recall that we are assuming a reasonable institution.)

Definition 35 Let $\varphi\colon \Sigma \to \Sigma'$ be a signature morphism and let $T = (\Sigma, E)$ be a Σ-theory. Then we let $\varphi \star T = (\Sigma', (\varphi(E))^\bullet)$. $\Box$

Fact 36 Let $\varphi\colon \Sigma \hookrightarrow \Sigma'$ be an inclusion and let $T = (\Sigma, E)$ be a Σ-theory. Then $\varphi \star T = (\Sigma', E^\bullet)$.

Proof: Because $Sen(\varphi)\colon Sen(\Sigma) \to Sen(\Sigma')$ is an inclusion, we have that $\varphi(e) = e$ for all $e \in Sen(\Sigma)$, and hence $\varphi(E) = E$. $\Box$

Proposition 37 Let T and T' be theories, and let φ and ψ be signature morphisms. Then:

1. If $\varphi\colon \Sigma(T) \to \Sigma'$ and $\psi\colon \Sigma' \to \Sigma''$, then $\psi \star (\varphi \star T) = (\varphi; \psi) \star T$.

2. If $i\colon \Sigma(T) \hookrightarrow \Sigma''$, $i'\colon \Sigma(T') \hookrightarrow \Sigma''$, $j\colon \Sigma(T) + \Sigma(T') \hookrightarrow \Sigma''$ and $\varphi\colon \Sigma'' \to \Sigma'''$, then $j; \varphi \star (T + T') = (i; \varphi \star T) + (i'; \varphi \star T')$. (One might write this loosely as $\varphi \star (T + T') = (\varphi \star T) + (\varphi \star T')$.)

Proof: First, let T be (Σ, E) and T' be (Σ', E'). Then

$$\psi \star (\varphi \star T) = \psi \star (\Sigma', (\varphi(E))^\bullet) = (\Sigma'', \psi((\varphi(E))^\bullet)^\bullet) = (\Sigma'', (\psi(\varphi(E))^\bullet)) = (\varphi; \psi) \star T,$$

where the second step follows by noting that the Satisfaction Condition implies that any consequence of $\varphi(E)$ is mapped by ψ into a consequence of $\psi(\varphi(E))$.

For the second assertion, we will rewrite both sides of the equation to the same expression:

$$j; \varphi \star (T + T') = (\Sigma''', (\varphi(E \cup E')^\bullet)^\bullet = (\Sigma''', (\varphi(E \cup E'))^\bullet)$$

and

$$(i; \varphi \star T) + (i'; \varphi \star T') = (\Sigma''', (\varphi E)^\bullet) + (\Sigma''', (\varphi E')^\bullet) = (\Sigma''', ((\varphi E)^\bullet \cup (\varphi E')^\bullet)^\bullet)$$
$$= (\Sigma''', (\varphi(E \cup E'))^\bullet).$$

$\Box$

3.3 Information Hiding

Information hiding is an important technique in modern programming, as well as in algebraic specification. Parnas [33] emphasized the importance of hiding implementation details within a module, in order to make it possible (for example) to improve a given data representation without having to search through all of a large program for each place where details of the representation are used. This is accomplished by *hiding* the data representation, i.e., by allowing access to it only through operations exported by its module. Similarly, Majster [30] showed that certain Σ-algebras cannot be specified as the initial Σ-algebra of a finite set of Σ-equations, while later work by Bergstra and Tucker (see [3], or a summary of related research in [32]) showed that any recursive Σ-algebra could be specified as the Σ-restriction of an initial Σ'-algebra of a finite set of Σ'-equations. Thus, there are some interesting Σ-theories that do not have finite Σ-presentations, but that are Σ-restrictions of finitely presented Σ'-theories, for some $\Sigma \subseteq \Sigma'$. In both cases, we have a signature inclusion $\Sigma \hookrightarrow \Sigma'$, and a Σ-theory $\Sigma \Box T$ of what is visible, derived by restricting a Σ'-theory T that includes both the visible and the hidden features. (Recall that we are assuming a reasonable institution.)

Definition 38 Let Σ' be a signature and let $T = (\Sigma, E)$ be a theory. Then

$$\Sigma' \Box T = (\Sigma \cap \Sigma', E \cap Sen(\Sigma')),$$

where $\Box$ is called the **export**, or **information hiding**, **operator**. $\Box$

Proposition 39 Given a signature inclusion $\Sigma' \hookrightarrow \Sigma$ and a Σ-theory T, then $\Sigma' \Box T$ is a Σ'-theory.

Proof: Let e be a Σ'-sentence such that $E \cap Sen(\Sigma') \models_{\Sigma'} e$. Then we have to prove that $e \in E \cap Sen(\Sigma')$. To prove that $e \in E$, it suffices to prove that $E \models_\Sigma e$. Let M be a model of E. Then also $M \models_\Sigma E \cap Sen(\Sigma')$. By the Satisfaction Condition, the reduct $M^{-}{}_{\Sigma'}$ also satisfies $E \cap Sen(\Sigma')$. It follows that $M^{-}{}_{\Sigma'} \models_{\Sigma'} e$. Now going backwards with the Satisfaction Condition, we obtain that $M \models_\Sigma e$, which proves this result. $\Box$

Corollary 40 For any signature Σ' and any Σ-theory T, $\Sigma' \Box T$ is a $(\Sigma \cap \Sigma')$-theory. $\Box$

4 Distributive Laws

This section discusses three distributive laws for information hiding over the sum of theories. The strong distributive law is what we really wish were true, because it says that to prove something about the visible behaviour of a module, we do not need to know anything about what is invisible. Unfortunately, this law does not seem to be satisfied by any institution that is widely used in Computing Science. The middle distributive law seems to be the strongest law that does hold in some institutions of interest; it is a restriction of the strong law by some conditions on the signatures involved. The weak distributive law is so weak that it is satisfied by almost all institutions that are widely used in Computing Science.

Definition 41 Let $\mathcal{I}$ be a reasonable institution. Then $\mathcal{I}$ satisfies the **strong distributive law** iff given a Σ_1-theory T_1, a Σ_2-theory T_2, and a signature Σ' such that $\Sigma' \hookrightarrow \Sigma_1$ and $\Sigma' \hookrightarrow \Sigma_2$, then

$$\Sigma' \Box (T_1 + T_2) = (\Sigma' \Box T_1) + (\Sigma' \Box T_2).$$

$\mathcal{I}$ satisfies the (**middle**) **distributive law** iff given a Σ_1-theory T_1 and a Σ_2-theory T_2, then

$$(\Sigma_1 \cap \Sigma_2)\Box(T_1 + T_2) = ((\Sigma_1 \cap \Sigma_2)\Box T_1) + ((\Sigma_1 \cap \Sigma_2)\Box T_2).$$

$\mathcal{I}$ satisfies the **weak distributive law** iff given a Σ_1-theory T_1 and signatures Σ_2 and Σ' such that $\Sigma' \hookrightarrow \Sigma_1$ and $\Sigma' \hookrightarrow \Sigma_2$, then

$$\Sigma'\Box(T_1 + T(\Sigma_2)) = (\Sigma'\Box T_1) + (\Sigma'\Box T(\Sigma_2)).$$

$\Box$

Notice that a reasonable institution satisfies the (middle) distributive law iff it satisfies the strong distributive law in the particular case where Σ' is $\Sigma(T_1) \cap \Sigma(T_2)$. Also, a reasonable institution satisfies the weak distributive law iff it satisfies the strong distributive law in the special case where one of the theories is the empty theory.

The following counterexample shows that equational logic does not satisfy the strong distributive law. Actually, this example applies to any reasonable institution that supports an internalisation for equality and for constants. This includes not only first order logic with equality, but also ordinary first order logic, by introducing a binary relational symbol for equality with sentences expressing its transitivity and symmetry.

Example 42 Let Σ' be an unsorted signature with just the two constants 1 and 2, and let Σ_1 and Σ_2 be Σ' with an additional constant 0. Let T_1 be the Σ_1-theory generated by the equation $1 = 0$, and let T_2 be the Σ_2-theory generated by the equation $2 = 0$. Then $\Sigma'\Box T_1$ and $\Sigma'\Box T_2$ are both empty, and so their sum is also empty. But $T_1 + T_2$ contains the equation $1 = 2$, which is also contained in $\Sigma'\Box(T_1 + T_2)$ because it is a Σ'-sentence. $\Box$

4.1 Distributive Laws and Interpolation

There are many different formulations of the Craig Interpolation Property at the level of institutions. Some that are equivalent for first order logic differ for many other important institutions, and some are so strong that they are not satisfied even by first order logic. We first formulate the Craig Interpolation Property for institutions in a style used by [37] for the equational case. Then we connect the middle distributive law with Craig Interpolation, in the style of [2] and [36].

Definition 43 A reasonable institution satisfies the **Craig Interpolation Property** iff for any signatures Σ and Σ' and any set E of Σ-sentences and any set E' of Σ'-sentences, if $E \models_{\Sigma+\Sigma'} E'$ then there is a set I of $(\Sigma \cap \Sigma')$-sentences such that $E \models_{\Sigma} I$ and $I \models_{\Sigma'} E'$. The set I is called the **interpolant** of E and E'. $\Box$

Without axiomatising inclusions, it is not possible to formulate Craig Interpolation in quite such an intuitive way. For example, Tarlecki [43] gives a more restrictive definition, involving a pushout of arbitrary signature morphisms. Another difference is that we consider *sets* of sentences rather than just single sentences. We agree with Rodenburg [37] that this is more natural; in particular, note that equational logic satisfies Definition 43, but not the single sentence version given in [43]. However, in an institution with (arbitrary) conjunctions, the single sentence and the set of sentence forms are equivalent.

Maibaum and Sadler [28] give a version of Craig Interpolation that is stronger in a different way, and Maibaum and Fiadeiro [29] give a precise formulation of it for π-institutions without inclusions. The following formulates this notion for inclusive institutions:

Definition 44 Let (Σ_1, E_1) and (Σ_2, E_2) be theories in an inclusive institution $\mathcal{I}$. Then $\mathcal{I}$ has the **Strong Craig Interpolation Property** iff given a Σ'-sentence e such that $E_1 \cup E_2 \models_{\Sigma_1+\Sigma_2} e$, there is a set I of $\Sigma_1 \cap \Sigma_2$ sentences such that $E_1 \models_{\Sigma_1} I$ and $E_2 \cup I \models_{\Sigma_2} e$. $\Box$

In order to state the relationships among Craig Interpolation, Strong Craig Interpolation, and the distributive law, we need the following:

Definition 45 An institution **is closed under implication** iff for any signature Σ, for any finite set E of Σ-sentences and any single Σ-sentence e, there is a set E' of Σ-sentences such that a Σ-model M satisfies E' iff it satisfies e whenever it satisfies E. We may write $(E \Rightarrow e)$ for this set E'. An institution **has false** iff for any signature Σ, there is a Σ-sentence False_{Σ} such that no Σ-model satisfies it. $\Box$

There are several ways to formulate results like the following, depending for example on whether closure under finite or arbitrary implication is assumed; but all proofs would be similar to that given below.

Proposition 46 If an institution $\mathcal{I}$ is compact and closed under implication, then it satisfies Craig Interpolation iff it satisfies Strong Craig Interpolation.

Proof: Craig Interpolation is the special case of Strong Craig Interpolation with $E_2 = ?$, so we only have to show that a compact institution with implication and Craig Interpolation satisfies Strong Craig Interpolation. Thus, assume that we are given theories (Σ_1, E_1) and (Σ_2, E_2), and let e be a Σ_2-sentence such that $E_1 \cup E_2 \models_\Sigma e$, where $\Sigma = \Sigma_1 \cup \Sigma_2$. By compactness, there are finite sets $E_1' \subseteq E_1$ and $E_2' \subseteq E_2$ such that $E_1' \cup E_2' \models_\Sigma e$, and by closure under implication, there is a set $(E_2' \Rightarrow e)$ of Σ_2-sentences such that for any Σ_2-model M, we have $M \models_{\Sigma_2} E_2$ implies $M \models_{\Sigma_2} e$ iff $M \models_{\Sigma_2} (E_2' \Rightarrow e)$. Then $E_1' \models_\Sigma (E_2' \Rightarrow e)$, because $E_1' \cup E_2' \models_\Sigma e$, by applying the Satisfaction Condition for the inclusion $\Sigma_2 \hookrightarrow \Sigma$. Craig Interpolation now gives us a set I of $\Sigma_1 \cap \Sigma_2$-sentences such that $E_1' \models_{\Sigma_1} I$ and $I \models_{\Sigma_2} (E_2' \Rightarrow e)$. Thus, $E_1' \models_{\Sigma_1} I$ and $I \cup E_2' \models_{\Sigma_2} e$, which implies that $E_1 \models_{\Sigma_1} I$ and $I \cup E_2 \models_{\Sigma_2} e$. $\Box$

Theorem 47 Let $\mathcal{I}$ be a reasonable compact institution closed under implication. Then $\mathcal{I}$ satisfies the distributive law if it satisfies the Craig Interpolation Property. Moreover, if $\mathcal{I}$ is a reasonable compact institution closed under implication and false, then it satisfies the distributive law iff it satisfies the Craig Interpolation Property.

Proof: Let $T_1 = (\Sigma_1, E_1)$ and $T_2 = (\Sigma_2, E_2)$ be theories. Let Σ' be $\Sigma(T_1) \cap \Sigma(T_2)$. We now calculate the two sides of distributive law. First,

$$\Sigma' \Box (T_1 + T_2) = \Sigma' \Box (\Sigma_1 + \Sigma_2, (E_1 \cup E_2)^\bullet) = (\Sigma', Sen(\Sigma') \cap (E_1 \cup E_2)^\bullet),$$

which is a theory by Proposition 39. Next,

$$(\Sigma' \Box T_1) + (\Sigma' \Box T_2) = (\Sigma', E_1 \cap Sen(\Sigma')) + (\Sigma', E_2 \cap Sen(\Sigma')) = (\Sigma', ((E_1 \cup E_2) \cap Sen(\Sigma'))^\bullet).$$

Now let E be the set of $(\Sigma_1 + \Sigma_2)$-sentences which is the set-theoretic union of E_1 and E_2. Notice that because $Sen(\Sigma') \cap E^\bullet$ is a theory (by Proposition 39 again) and because $E \cap Sen(\Sigma')$ is included in it, we have $(E \cap Sen(\Sigma'))^\bullet \subseteq Sen(\Sigma') \cap E^\bullet$. We will show that the opposite inclusion follows from the Craig Interpolation Property when $\mathcal{I}$ is compact and closed under implication.

We have to prove that for any $(\Sigma_1 \cap \Sigma_2)$-sentence e', if $E_1 \cup E_2 \models_{\Sigma_1+\Sigma_2} e'$ then $(E_1 \cup E_2) \cap Sen(\Sigma') \models_{\Sigma'} e'$. By compactness, there are finite sets of sentences $E_1^0 \subseteq E_1$ and $E_2^0 \subseteq E_2$ such that $E_1^0 \cup E_2^0 \models_{\Sigma_1+\Sigma_2} e'$. $(E_2^0 \Rightarrow e')$ is a Σ_2-sentence by construction, and $E_1^0 \models_{\Sigma_1+\Sigma_2} (E_2^0 \Rightarrow e')$.

Now let $I \subseteq Sen(\Sigma')$ be an interpolant such that $E_1^0 \models_{\Sigma_1} I \models_{\Sigma_2} (E_2^0 \Rightarrow e')$. By compactness, we may assume that I is finite.

From $E_1 \models_{\Sigma_1} I$ it follows that $I \subseteq E_1$, because E_1 is closed. Thus $I \subseteq E_1 \cap Sen(\Sigma')$. Also, $I \models_{\Sigma_2} (E_2^0 \Rightarrow e')$ implies that $E_2^0 \models_{\Sigma_2} (I \Rightarrow e')$, which implies that $(I \Rightarrow e') \subseteq E_2$, because E_2 is closed. Therefore $(I \Rightarrow e') \subseteq E_2 \cap Sen(\Sigma')$.

The desired conclusion now follows from the facts that $I \cup (I \Rightarrow e') \subseteq (E_1 \cup E_2) \cap Sen(\Sigma')$ and that $I \cup (I \Rightarrow e') \models_{\Sigma'} e'$.

Now for the converse, suppose that $\mathcal{I}$ also has false and satisfies the distributive law. Because $\mathcal{I}$ is compact, the Craig Interpolation Property is equivalent to its finitary version (in which all of the sets of sentences in the definition of the Craig Interpolation Property are finite).

Consider finite sets E_1 of Σ_1-sentences and E_2 of Σ_2-sentences such that $E_1 \models_{\Sigma_1+\Sigma_2} E_2$. Let T_1 be the Σ_1-theory generated by E_1 and T_2 the Σ_2-theory generated by $(E_2 \Rightarrow \mathit{False}_{\Sigma_2})$. Because $\mathit{False}_{\Sigma'} = \mathit{False}_{\Sigma_1+\Sigma_2}$ lies in $T_1 + T_2$, it follows that $\mathit{False}_{\Sigma'}$ belongs to $\Sigma' \Box T_1 + \Sigma' \Box T_2$.

There are finite sets of sentences E_1^0 in $\Sigma' \Box T_1$ and E_2^0 in $\Sigma' \Box T_2$ such that $E_1^0 \cup E_2^0 \models_{\Sigma'} \mathit{False}$. We claim that E_1^0 will serve as an interpolant. Because T_1 is generated by E_1 we have that $E_1 \models_{\Sigma_1} E_1^0$. From $E_1^0 \cup (E_2 \Rightarrow \mathit{False}) \models_{\Sigma'} \mathit{False}$, it follows that $E_1^0 \models_{\Sigma_2} E_2$. $\Box$

It may be worth noting that in proving this result, we could assume closure under arbitrary implication, instead of compactness and closure under finite implication. But this would be less realistic, because computing systems only handle finite data.

There is an apparently stronger formulation of the Middle Distributive Law that is actually equivalent:

Theorem 48 A distributive inclusive institution with Craig Interpolation satisfies the (middle) distributive law iff for every Σ_1-theory T_2, Σ_2-theory T_2, and signature Σ,

$$(\Sigma_1 \cap \Sigma_2) \subseteq \Sigma \text{ implies } \Sigma \Box (T_1 + T_2) = (\Sigma \Box T_1) + (\Sigma \Box T_2).$$

We will call this property the **strong middle distributive law**.

Proof: The distributive law is the special case where $\Sigma = \Sigma_1 \cap \Sigma_2$, so we only need to show that the strong form follows from the weak form.

Let Σ_1, Σ_2 and Σ be signatures such that $\Sigma_1 \cap \Sigma_2 \subseteq \Sigma$, and let (Σ_1, E_1) and (Σ_2, E_2) be theories. Then we must prove that

$$\Sigma \Box (\Sigma_1, E_1) + \Sigma \Box (\Sigma_2, E_2) = \Sigma \Box ((\Sigma_1, E_1) + (\Sigma_2, E_2)).$$

A simple computation shows that the signature of each side of this equality is $\Sigma \cap (\Sigma_1 + \Sigma_2)$, which we will denote Σ'. The equality can be therefore rewritten as

$$(\Sigma', (Sen(\Sigma') \cap (E_1 \cup E_2))^\bullet) = (\Sigma', Sen(\Sigma') \cap (E_1 \cup E_2)^\bullet).$$

Because the lefthand side is clearly included in the righthand side, we need only prove the opposite inclusion, that is, if $E_1 \cup E_2 \models_{\Sigma_1+\Sigma_2} e$ for $e \in Sen(\Sigma')$, then $Sen(\Sigma') \cap (E_1 \cup E_2) \models_{\Sigma'} e$. Let $\Sigma_1' = \Sigma_1 + (\Sigma \cap \Sigma_2)$ and let $\Sigma_2' = \Sigma_2 + (\Sigma \cap \Sigma_1)$. Then $\Sigma_1' \cap \Sigma_2' = (\Sigma_1 + (\Sigma \cap \Sigma_2)) \cap (\Sigma_2 + (\Sigma \cap \Sigma_1)) =$

$(\Sigma+\Sigma_1)\cap(\Sigma_1+\Sigma_2)\cap(\Sigma+\Sigma_2)\cap(\Sigma_1+\Sigma_2) = (\Sigma+(\Sigma_1\cap\Sigma_2))\cap(\Sigma_1+\Sigma_2)$. Because $\Sigma_1\cap\Sigma_2\subseteq\Sigma$, we have that $\Sigma'_1\cap\Sigma'_2 = \Sigma\cap(\Sigma_1+\Sigma_2) = \Sigma'$.

Now let E'_1 be the Σ'_1-theory generated by E_1, let E'_2 be the Σ'_2-theory generated by E_2, and suppose that $E_1\cup E_2 \models_{\Sigma_1+\Sigma_2} e$ with $e\in Sen(\Sigma')$. Because $E_1\subseteq E'_1$ and $E_2\subseteq E'_2$, we also have that $E'_1\cup E'_2\models_{\Sigma_1+\Sigma_2} e$. Now we are in a position to apply the middle distributive law for (Σ'_1, E'_1) and (Σ'_2, E'_2), which gives $Sen(\Sigma')\cap(E'_1\cup E'_2)\models_{\Sigma'} e$.

Next we show that $Sen(\Sigma')\cap(E_1\cup E_2)\models_{\Sigma'} E'_1\cap Sen(\Sigma')$. Pick any Σ'-sentence e' in E'_1. Because E'_1 is the Σ'_1-theory generated by E_1, we have that $E_1\models_{\Sigma'_1} e'$. But $\Sigma'_1 = \Sigma_1+(\Sigma\cap\Sigma_2) = \Sigma_1+\Sigma'$, while E_1 is a set of Σ_1-sentences, and e' is a Σ'-sentence. So by the Craig Interpolation Property, we get $I\subseteq Sen(\Sigma'\cap\Sigma_1) = Sen(\Sigma')\cap Sen(\Sigma_1)$ such that $E_1\models_{\Sigma_1} I$ and $I\models_{\Sigma'} e'$. Because E_1 is a theory and $I\subseteq Sen(\Sigma_1)$, it follows that $I\subseteq E_1$. Because $I\subseteq Sen(\Sigma')$ as well, $I\subseteq E_1\cap Sen(\Sigma')$. Thus, $E_1\cap Sen(\Sigma')\models_{\Sigma'} e'$, which means that $(E_1\cup E_2)\cap Sen(\Sigma')\models_{\Sigma'} e'$.

Similarly, we obtain that $(E_1\cup E_2)\cap Sen(\Sigma')\models_{\Sigma'} E'_2\cap Sen(\Sigma')$.

Therefore $(E_1\cup E_2)\cap Sen(\Sigma')\models_{\Sigma'} (E'_1\cup E'_2)\cap Sen(\Sigma')\models_{\Sigma'} e$. □

Corollary 49 In a distributive inclusive institution that is closed under implication and has false, the followings are equivalent:

1. the Craig Interpolation Property;
2. the middle distributive law;
3. the strong middle distributive law.

Proof: This follows from Theorem 48 and Theorem 47. □

The proof of Theorem 47 resembles the proof of equivalence of the distributive law and the Craig Interpolation Property given in [2] for first order logic. Indeed, we can apply the above result to obtain the distributive law for first order logic, because first order logic is compact, closed under implication and false, and satisfies the Craig Interpolation Property. However, equational logic, and also conditional equational logic, are not closed under implication and false, so that we cannot obtain the distributive law in this way for these logical systems. In fact, the distributive law does not hold for them, as shown by the following counterexample, adapted from [2]:

Example 50 Consider the unsorted signatures Σ_1 containing the constants 0, c_1 and c_2 and the unary operation symbol $-$, and Σ_2 containing the constants c_1 and c_2, the unary operation symbol $-$, and the ternary operation symbol h.

Let T_1 be the Σ_1-theory generated by the equation $-0 = 0$ and and let T_2 be the Σ_2-theory generated by the equations $h(x,x,y) = y$, $h(x,-x,c_1) = h(x,-x,c_2)$ and $-(-x) = x$. The signature $\Sigma = \Sigma_1\cap\Sigma_2$ contains only c_1 and c_2 as constants and the unary operation symbol $-$. T_1+T_2 contains the equation $c_1 = c_2$ and so does $\Sigma\Box(T_1+T_2)$. Notice that $\Sigma\Box T_1$ is empty and that $\Sigma\Box T_2$ does not contain $c_1 = c_2$. Therefore $c_1 = c_2$ does not belong to $(\Sigma\Box T_1)+(\Sigma\Box T_2)$. □

A simpler example could be given along the lines of the example in [19] showing the need for explicit quantification in many sorted equational logic with possibly empty carriers. However, the example given above is stronger, because it shows the (middle) distributive law does not hold for unsorted equational logic. We will see in Section 4.3 that this law does hold when the theory inclusions are conservative extensions, and we will argue that this covers the expected applications.

4.2 The Weak Distributive Law

This subsection shows that the weak distributive law holds for all of the institutions that are generally used for specification in Computing Science, by reducing it to a condition that is easy to check.

Definition 51 An inclusive institution $\mathcal{I}$ is **uniform** iff for any inclusion of signatures $\Sigma' \hookrightarrow \Sigma$, any set E' of Σ'-sentences, and any Σ'-sentence e', if $E' \models_\Sigma e'$ then $E' \models_{\Sigma'} e'$. $\Box$

Intuitively, this condition says that any semantic consequence e also follows using only the symbols that actually occur in it. Also, note that we can apply this definition to a syntactic consequence relation by using Theorem 33 of Section 2.4.

Proposition 52 An inclusive institution satisfies the weak distributive law iff it is uniform.

Proof: Consider a signature Σ' which is included in both Σ_1 and Σ_2. Let E_2 be a theory over Σ_2. Now we calculate the two sides of the weak distributive law. First,

$$\Sigma'\Box((\Sigma_1, ?^\bullet) + (\Sigma_2, E_2)) = \Sigma'\Box(\Sigma_1 + \Sigma_2, E_2^\bullet) = (\Sigma', Sen(\Sigma') \cap E_2^\bullet),$$

which is a theory by Proposition 39; notice that the closure of E_2 is taken in $\Sigma_1 + \Sigma_2$. Next,

$$(\Sigma'\Box(\Sigma_1, ?^\bullet)) + (\Sigma'\Box(\Sigma_2, E_2)) = (\Sigma', ?^\bullet) + (\Sigma', E_2 \cap Sen(\Sigma')) = (\Sigma', E_2 \cap Sen(\Sigma')).$$

Clearly, we have the inclusion $E_2 \cap Sen(\Sigma') \subseteq Sen(\Sigma') \cap E_2^\bullet$.

We next show that the opposite inclusion is equivalent to uniformity. Suppose that uniformity holds for the inclusion $\Sigma_2 \hookrightarrow \Sigma_1 + \Sigma_2$. Pick a Σ_2-sentence e' such that $E_2 \models_{\Sigma_1+\Sigma_2} e'$. Then $E_2 \models_{\Sigma_2} e'$, which implies that $Sen(\Sigma_2) \cap E_2^\bullet \subseteq E_2$. The desired inequality follows by intersecting this inequality with $Sen(\Sigma')$.

The converse follows from the particular case where $\Sigma_2 \hookrightarrow \Sigma_1$. The weak distributive law (or more precisely, the inclusion mentioned above) now gives the uniformity condition for $\Sigma_2 \hookrightarrow \Sigma_1$. $\Box$

All logical system that are widely used in Computing Science are uniform (under the usual notion of inclusion). The rest of this subsection gives some machinery for establishing the uniformity of institutions.

Definition 53 An inclusive institution **has model expansion** iff for every signature inclusion $\Sigma' \hookrightarrow \Sigma$ and for every Σ'-model M' there is a Σ-model M such that $M\!\upharpoonright_{\Sigma'} = M'$. $\Box$

In the applications, this property is obtained by interpreting the symbols of Σ that are not in Σ' in an arbitrary way in the model M'. This can be done only if we do not have empty carriers in the models. This can be assured by assuming the existence of at least one constant for any sort which is the target of a new operation symbol in Σ'. Model expansion is used, for example, by Rodenburg [36] in his elegant proof of the Craig Interpolation Property for conditional equational logic.

Lemma 54 Any inclusive institution with model expansion is uniform.

Proof: Let $\Sigma' \hookrightarrow \Sigma$ be an inclusion of signatures. Let E' be a set of Σ'-sentences and let e' be a single Σ'-sentence such that $E' \models_\Sigma e'$. Our aim is to prove that $E' \models_{\Sigma'} e'$.

Pick a Σ'-model M' of E'. Let M be a Σ-model such that $M\!\upharpoonright_{\Sigma'} = M'$. By the Satisfaction Condition, $M \models_\Sigma E'$. It follows that $M \models_\Sigma e'$. Going backwards with the Satisfaction Condition, we obtain $M' \models_{\Sigma'} e'$. This implies that $E' \models_{\Sigma'} e'$. $\Box$

Corollary 55 An inclusive institution satisfies the weak distributive law if it has model expansion. □

Thus, the weak distributive law is closely related to model expansion and uniformity, and is not very closely related to the Craig Interpolation Property.

4.3 The Distributive Law and Conservative Extensions

We have seen that the (middle) distributive law does not hold for equational logics. This subsection shows that it does hold for semiexact reasonable institutions, including equational logics, when only conservative extensions are used. We argue that conservative extensions are the only case of real interest for applications. This follows from noting that good design practice demands that an imported subsystem should not exhibit behaviour in its new context that differs from what its specification says it should do, i.e., the inclusion of a module into a larger system should be conservative. (If some specification imports a module in such a way that its behaviour is changed, then it should use a *copy* of that module, so as to disconnect it from the original specification. Syntax and semantics for this are provided, for example, in the OBJ3 language; see [8, 23].)

Definition 56 A theory morphism $\varphi\colon (\Sigma, E) \to (\Sigma', E')$ is **conservative** iff for each (Σ, E)-model M there is a (Σ', E')-model M' such that M is the φ-reduct of M' (i.e., $M'^{-}{}_{\varphi} = M$). Also φ is a **conservative extension** iff it is an extension and is conservative. □

Theorem 57 In a semiexact reasonable institution, if $T_1 = (\Sigma_1, E_1)$ and $T_2 = (\Sigma_2, E_2)$ are theories, and if the extensions $(\Sigma_1 \cap \Sigma_2)\Box T_1 \hookrightarrow T_1$ and $(\Sigma_1 \cap \Sigma_2)\Box T_2 \hookrightarrow T_2$ are conservative, then the distributive law holds for the theories T_1 and T_2.

Proof: Let Σ' denote $\Sigma_1 \cap \Sigma_2$. By the computations in the proof of Theorem 47, we see that the distributive law for T_1 and T_2 is equivalent to

$$(\Sigma', Sen(\Sigma') \cap (E_1 \cup E_2)^\bullet) \subseteq (\Sigma', ((E_1 \cup E_2) \cap Sen(\Sigma'))^\bullet).$$

Let e' be a Σ'-sentence such that $E_1 \cup E_2 \models_{\Sigma_1+\Sigma_2} e'$ and pick a Σ'-model M' of $Sen(\Sigma') \cap (E_1 \cup E_2)$. Because M' is a model for $Sen(\Sigma') \cap E_1$ and because $\Sigma'\Box T_1 \hookrightarrow T_1$ is conservative, there is a Σ_1-model M_1 that satisfies E_1 and whose reduct $M_1{}^{-}{}_{\Sigma'}$ is M'. Similarly, there is a Σ_2-model M_2 that satisfies E_2 and whose reduct $M_2{}^{-}{}_{\Sigma'}$ is M'.

Because *Mod* preserves pushouts and $(\Sigma_1 + \Sigma_2)$ is the pushout of $\Sigma' \hookrightarrow \Sigma_1$ and $\Sigma' \hookrightarrow \Sigma_2$ (by Proposition 11), $Mod(\Sigma_1 + \Sigma_2)$ is the pullback of $Mod(\Sigma_1) \to Mod(\Sigma')$ and $Mod(\Sigma_2) \to Mod(\Sigma')$. Then there is a $(\Sigma_1 + \Sigma_2)$-model M such that $M^{-}{}_{\Sigma_1} = M_1$ and $M^{-}{}_{\Sigma_2} = M_2$. By the Satisfaction Condition, $M \models_{\Sigma_1+\Sigma_2} E_1$ and $M \models_{\Sigma_1+\Sigma_2} E_2$. Therefore $M \models_{\Sigma_1+\Sigma_2} e'$. Finally, because $M' = M^{-}{}_{\Sigma}$, we conclude that $M' \models_{\Sigma} e'$. □

(This paragraph is for those interested in the implementation of equational languages: Theorem 57 implies that one could use a technique related to the implementation of OBJ [23, 8], which treats different kinds of extensions in different ways. For conservative extensions, the component theories may be summed in the usual way, with sharing of common subtheories; but for non-conservative extensions, the component theories should be copied, so that new sentences about component theories do not pollute these old theories.)

4.4 Summary Concerning the Distributive Laws

Let us now summarize the situation for the distributive laws. The condition that one might really wish for, the strong distributive law, holds in none of the institutions that are generally used for specification in Computing Science. This corresponds to the intuition and experience of designers and users, that not all of the properties of a complex system that are visible through some interface are explicable without reference to the internals of the system. However, a weaker version of the distributive law, here called the middle distributive law, does hold for some institutions of interest. In our view, this is not a particularly significant property, because its hypothesis unrealistically requires the external visibility of all internal interfaces of a system. Note that it is not in general equivalent to the Craig Interpolation Property; for example, equational logic satisfies Craig interpolation but not the middle distributive law. However, these two conditions are equivalent for institutions with sufficiently rich resources for combining sentences, and they are equivalent to a strong form of the middle distributive law. Also, at least for first order logic, the middle distributive law supports a normal form for module expressions, in which information hiding can be reduced to a single operation which is performed last [2]. The last subsection showed that the middle distributive law holds for equational style logics when the extensions are conservative, and argued that this is the only case of real interest in applications. Finally, let us recall that the weak distributive law is so weak that it holds in all of the institutions that are widely used in Computing Science.

5 Pushouts of Extensions

The concept of conservative extension can be defined either semantically or syntactically, and in general the syntactic definition is strictly stronger. Byers and Pitt [6] give a nice discussion, which includes some counterexamples; Veloso and Veloso [47] discuss such examples in more detail for the case of first order logic. In Computing Science applications, models are specific programs running on specific machines, or specific VLSI chips running in particular contexts, and are the objects of primary interest. Therefore we feel that the model theoretic definition of conservative extension should be given priority. The fact that the syntactic formulation does not exactly characterize what happens to the models seems to us evidence against a deduction-based approach to the foundations of software engineering, as advocated by Maibaum, Turski, Sadler and others [28, 46]. However, this does not mean that π-institutions are useless; for example, they are still appropriate for their original application to deductive databases [7], where indeed we may say that theories serve as models, as in Theorem 33.

This section considers pushouts of three different variants of the notion of conservative extension. The first of these is the notion in Definition 56 from the previous section. Then we consider a stronger form of conservatism called persistency, that is relevant to parameterized modules. Finally, we consider extensions that conserve initial models of the theories involved; this is relevant to the specification of data types.

5.1 Conservative Extensions

This section shows that pushouts of conservative extensions are conservative, under some minimal assumptions.

Proposition 58 Given a semiexact institution with pushouts of signatures, let (φ_1', φ_2') be a pushout of theory morphisms $\varphi_1 : T' \to T_1$ and $\varphi_2 : T' \to T_2$. Then $\varphi_1' : T_2 \to T$ is conservative if $\varphi_1 : T' \to T_1$ is conservative.

Proof: Suppose that $\varphi_1 : T' \to T_1$ is conservative, and pick an arbitrary model M_2 of T_2. Then $M_2{}^{-}{}_{\varphi_2}$ is a model of T' by the Satisfaction Condition. Because φ_1 is conservative, there is a model M_1 of T_1 such that $M_1{}^{-}{}_{\varphi_1} = M_2{}^{-}{}_{\varphi_2}$. By Proposition 28, there is a model M of T such that $M^{-}{}_{\varphi_2'} = M_1$ and $M^{-}{}_{\varphi_1'} = M_2$. Because M_2 was arbitrarily chosen, we conclude that any T_2-model has an φ_1'-expansion to a T-model. □

Corollary 59 In any semiexact inclusive institution in which pushouts of signatures preserve inclusions, pushouts also preserve conservative extensions.

Proof: We can lift the preservation of inclusions by pushouts from signatures to theories by Fact 30, and then apply Proposition 58. □

The following notion is often found in the literature. We will show that it is a necessary but insufficient condition for conservation:

Definition 60 A theory extension $(\Sigma, E) \hookrightarrow (\Sigma', E')$ is **syntactically conservative** iff $E = E' \cap Sen(\Sigma)$. □

Maibaum and Sadler [28] give an example showing that this notion of conservatism is not preserved under pushout.

Proposition 61 If a theory extension $(\Sigma, E) \hookrightarrow (\Sigma', E')$ is conservative, then it is syntactically conservative.

Proof: It suffices to show that $E' \cap Sen(\Sigma) \subseteq E$, because $E \subseteq E'$ already. So let $e' \in E' \cap Sen(\Sigma)$; then it suffices to show $e' \in E$.

Let $M \models_\Sigma E$. Then there is an M' such that $M \models_{\Sigma'} E'$ and $M'{}^{-}{}_\Sigma = M$. Therefore $M' \models_{\Sigma'} e'$. Thus, $M'{}^{-}{}_\Sigma \models_\Sigma e'$, i.e., $M \models_\Sigma e'$. Therefore $e' \in E^\bullet = E$. □

The example of Maibaum and Sadler [28] shows that syntactic conservatism is not preserved under pushouts in the equational institution. From this, we may conclude that syntactic conservatism is not a sufficient condition for true conservatism. For if it were, then Corollary 59 and the above proposition would imply that pushouts preserve syntactic conservativism in any semiexact inclusive institution, including equational logic.

5.2 Persistent Extensions

Persistence is a stronger notion than conservative extension, and is important for the semantics of parametrized data types.

Definition 62 A theory morphism $\varphi\colon T' \to T$ is **persistent** iff its associated reduct functor $\_{}^{-}{}_\varphi : Mod(T) \to Mod(T')$ has a left adjoint such that each component of the unit of the adjunction is an equality. □

Fact 63 A persistent theory morphism is conservative. □

The following result is related to the semantics of applying a generic module to an actual parameter module using a "view," as proposed in Clear and implemented in OBJ3:

Proposition 64 Given a semiexact institution with pushouts of signatures, let (φ_1', φ_2') be the pushout of theory morphisms $\varphi_1 : T' \to T_1$ and $\varphi_2 : T' \to T_2$. Then the pushout morphism $\varphi_1' : T_2 \to T$ is persistent if $\varphi_1 : T' \to T_1$ is persistent.

Proof: Suppose that the morphism $\varphi_1 : T' \to T_1$ is persistent, and pick an arbitrary model M_2 of T_2. Then $M_2{}^{-}{}_{\varphi_2}$ is a model of T' by the Satisfaction Condition. Let M_1 be the free T_1-model over $M_2{}^{-}{}_{\varphi_2}$. Then $M_1{}^{-}{}_{\varphi_1} = M_2{}^{-}{}_{\varphi_2}$ by the persistency of φ_1. By Proposition 28, there is a model M of T such that $M^{-}{}_{\varphi_2'} = M_1$ and $M^{-}{}_{\varphi_1'} = M_2$. We will show that M is the free T-model over M_2.

Pick an arbitrary T-model N and an arbitrary Σ_2-morphism $h_2 \colon M_2 \to N^{-}{}_{\varphi_1'}$. We have to show that there is an unique Σ-morphism $h \colon M \to N$ such that $h^{-}{}_{\varphi_1'} = h_2$. Notice that by Proposition 28, any $h \colon M \to N$ is uniquely determined by its reducts $h_2 = h^{-}{}_{\varphi_1'} : M_2 \to N^{-}{}_{\varphi_1'}$ and $h_1 = h^{-}{}_{\varphi_2'} : M_1 \to N^{-}{}_{\varphi_2'}$ and by the condition that $h_1{}^{-}{}_{\varphi_1} = h_2{}^{-}{}_{\varphi_2}$.

We let h_1 be the unique Σ_1-morphism $M_1 \to N^{-}{}_{\varphi_2'}$ such that $h_1{}^{-}{}_{\varphi_1} = h_2{}^{-}{}_{\varphi_2}$ (remember that M_1 is free over $M_2{}^{-}{}_{\varphi_2}$). Then the Σ-morphism $h \colon M \to N$ determined by (h_1, h_2) is the desired extension of h_2 to a Σ-morphism $M \to N$. $\Box$

Corollary 65 In any semiexact inclusive institution in which pushouts of signatures preserve inclusions, pushouts also preserve persistent extensions.

Proof: We can lift the preservation of inclusions by pushouts from signatures to theories by Fact 30, and then apply Proposition 64. $\Box$

5.3 Conservative for Initiality

A rather different notion of conservative extension is appropriate when considering data types that are defined by initial model semantics.

Definition 66 A theory morphism $T' \to T$ is **conservative for initiality** iff the theories T' and T admit initial models, $0_{T'}$ and 0_T, respectively, such that $0_T{}^{-}{}_{\Sigma'} = 0_{T'}$. $\Box$

Fact 67 Given a semiexact institution with pushouts of signatures, if both theories T' and T of a persistent morphism $h \colon T' \to T$ admit initial models, then h is conservative for initiality.

Proof: This follows from the preservation of initial objects (which are colimits) by left adjoint functors. $\Box$

Proposition 68 Given a semiexact institution with pushouts of signatures, let (φ_1', φ_2') be the pushout of theory morphisms $\varphi_1 : T' \to T_1$ and $\varphi_2 : T' \to T_2$. If both φ_1 and φ_2 are conservative for initiality, then the pushout morphisms φ_1' and φ_2' are also conservative for initiality.

Proof: Because $0_{T_1}{}^{-}{}_{\varphi_1} = 0_{T_2}{}^{-}{}_{\varphi_2}$ by Proposition 28, 0_{T_1} and 0_{T_2} determine a T-model 0_T such that $0_T{}^{-}{}_{\varphi_2'} = 0_{T_1}$ and $0_T{}^{-}{}_{\varphi_1'} = 0_{T_2}$. The initiality of 0_T follows from the initiality of 0_{T_1} and 0_{T_2} and the fact that any T-morphism $0_T \to M$ is uniquely determined by its reducts $0_{T_1} \to M^{-}{}_{\varphi_2'}$ and $0_{T_2} \to M^{-}{}_{\varphi_1'}$. $\Box$

6 Conclusions

In order to compare the properties of different logical systems, it is helpful to employ some formalisation of the notion of logical system. Otherwise, it is difficult to be sure that general properties have been formulated correctly, and it is impossible to rigorously compare relationships among different general properties. We have used institutions to formalize logical systems. An alternative formalisation would have been π-institutions. But these leave out the vital connection with models; also, a result in this paper shows that π-institutions can be seen as a special case of ordinary institutions.

Some authors have argued against equational logic as a foundation for software engineering on technical grounds, including:

1. the failure of pushouts to preserve conservative extensions [28];
2. failure of the (middle) distributive law for information hiding over sum [2]; and
3. failure of the Strong Craig Interpolation Property [28, 29].

The first point rests on using a *syntactic* formulation of conservative extension, rather than the *semantic* formulation that we have argued correctly describes the behaviour of the models (e.g., running programs, or chips) that are the result of the design process. Indeed, the fact that pushouts do not necessarily preserve syntactically conservative extensions, even for equational logic, seems to us further confirmation of the inappropriateness of that formulation.

As argued in Section 4.3, non-conservative importation of modules is of little practical interest, because good design practice demands that imported modules should behave as advertised in their new context. Therefore failure to satisfy the middle for all extensions should not be held against an institution.

The third point rests on a result relating the Strong Craig Interpolation Property to the preservation of conservative extensions under pushouts. However, we have argued above that the syntactic formulation of conservative extension used in this result is inappropriate; hence Strong Craig Interpolation is not needed.

In general, those attacking equational logic have preferred first order logic. But in fact, there are many desirable properties satisfied by equational logic that are not satisfied by classical first order logic, including the following:

1. the existence of **initial models** for all theories, and more pointedly, a close connection with the most natural models for modules, which are the **computable algebras** for data types, or more generally, semi-computable models.
2. **left adjoints** for all forgetful functors induced by theory morphisms (i.e., liberality), which supports **free extensions** of models; and
3. **algorithms** like term rewriting, Knuth-Bendix, and narrowing, which give equational logic a strongly computational flavour, and make it especially suitable for mechanisation.

Some additional points are that equational logic can be used conveniently for the specification and verification of imperative programs [18], and also object oriented programs [13]. Moreover, equational deduction is significantly simpler than deduction for full first order logic, and as noted above, has significant advantages for computation. Thus, the case for using equational logic in Computing Science seems quite strong. Of course, equational logic cannot be used for all applications. But we feel that it should be used wherever it can, including hardware and software specification, for the reasons given above. Also, because of the connection with charters and parchments developed in [15], equational logic provides a good metalanguage for describing other logics, and in particular, it seems promising as a metalanguage for theorem proving over arbitrary logical systems.

To try to sum up the whole paper now, we hope to have shown that institutions can be used to define a variety of properties of logical systems that support modularisation, and to clarify the relationships among them. In particular, we have shown that certain kinds of conservative extension are preserved by pushouts, under certain conditions. These results include extensions that are conservative for initiality, and that are persistent, as well as ordinary conservative extensions. Extensions that are conservative for initiality are important for importing (abstract) data types, and our result for this case requires that both extensions are conservative for initiality. The second result is important for applications of parameterised modules, while the third applies to importing and reusing modules. We have also considered a number of algebraic laws for operations on modules, and in particular, we have argued that the distributive laws for information hiding over the sum of theories are not particularly significant for Computing Science, except in the case where the theory extensions involved are conservative, and we have given a general theorem that covers this case. Most of our results involve assumptions of (semi)exactness and reasonability, which appear to hold for most institutions of interest in Computing Science. Finally, we have shown how to regard π-institutions as ordinary institutions, so that our results also apply to settings where deduction is given priority over models.

References

[1] Hajnal Andréka and István Németi. A general axiomatizability theorem formulated in terms of cone-injective subcategories. In *Universal Algebra*, pages 13–35. North-Holland, 1981. Colloquia Mathematics Societas János Bolyai, 29.

[2] Jan Bergstra, Jan Heering, and Paul Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.

[3] Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In *Automata, Languages and Programming, Seventh Colloquium*, pages 76–90. Springer, 1980. Lecture Notes in Computer Science, Volume 81.

[4] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[5] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland.

[6] Paddy Byers and David Pitt. Conservative extensions: A cautionary note. *Bulletin of the European Association for Theoretical Computer Science*, 41:196–201, June 1990.

[7] José Fiadeiro and Amilcar Sernadas. Structuring theories on consequence. In Donald Sannella and Andrzej Tarlecki, editors, *Recent Trends in Data Type Specification*, pages 44–72. Springer, 1988. Lecture Notes in Computer Science, Volume 332.

[8] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[9] Joseph Goguen. Types as theories. To appear in *Proceedings, Symposium on General Topology and Applications* (Oxford, June 1989) Oxford, 1991.

[10] Joseph Goguen. Categorical foundations for general systems theory. In *Advances in Cybernetics and Systems Research*, pages 121–130. Transcripta Books, 1973.

[11] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251-263, and in *Domain Analysis and Software Systems Modelling*, Ruben Prieto-Diaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125-137.

[12] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison-Wesley, 1989.

[13] Joseph Goguen. An algebraic approach to refinement. In *Proceedings, VDM'90: VDM and Z – Formal Methods in Software Development*, pages 12–28. Springer, 1990. Lecture Notes in Computer Science, Volume 428.

[14] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Technical Report CSL-118, SRI Computer Science Lab, October 1980.

[15] Joseph Goguen and Rod Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In *Proceedings, Conference on Category Theory and Computer Programming*, pages 313–333. Springer, 1986. Lecture Notes in Computer Science, Volume 240; also, Report Number CSLI-86-54, Center for the Study of Language and Information, Stanford University, June 1986.

[16] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, to appear. Also, Report ECS-LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; an early ancestor is "Introducing Institutions," in *Proceedings, Logics of Programming Workshop*, Edward Clarke and Dexter Kozen, editors, Springer Lecture Notes in Computer Science, Volume 164, pages 221-256, 1984.

[17] Joseph Goguen and Susanna Ginali. A categorical approach to general systems theory. In George Klir, editor, *Applied General Systems Research*, pages 257–270. Plenum, 1978.

[18] Joseph Goguen and Grant Malcolm. *Algebraic Semantics for Imperative Languages.* Draft, 1991.

[19] Joseph Goguen and José Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334, 1985. Preliminary versions have appeared in: *SIGPLAN Notices*, July 1981, Volume 16, Number 7, pages 24-37; SRI Computer Science Lab Technical Report CSL-135, May 1982; and Report CSLI-84-15, Center for the Study of Language and Information, Stanford University, September 1984.

[20] Joseph Goguen and José Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In Douglas DeGroot and Gary Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179-210, September 1984.

[21] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[22] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80-149.

[23] Joseph Goguen and Timothy Winkler. Introducing OBJ3. Technical Report SRI-CSL-88-9, SRI International, Computer Science Lab, August 1988. Revised version to appear with additional authors José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud in *Applications of Algebraic Specification using OBJ*, edited by Joseph Goguen, Derek Coleman and Robin Gallimore, Cambridge, 1990.

[24] Robert Goldblatt. *Topoi, the Categorial Analysis of Logic.* North-Holland, 1979.

[25] Robert Harper, Donald Sannella, and Andrzej Tarlecki. Logic representation in LF. In *Proceedings, Conference on Category Theory and Computer Science*, pages 250–272. Springer, 1989. Lecture Notes in Computer Science, Volume 389.

[26] Horst Herrlich and George Strecker. *Category Theory.* Allyn and Bacon, 1973.

[27] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer, 1971.

[28] Thomas Maibaum and Martin Sadler. Axiomatising specification theory. In *Recent Trends in Data Type Specification*, pages 171–177. Springer, 1985. Informatik-Fachberichte 116.

[29] Thomas Maibaum and José Fiadeiro with Martin Sadler. Stepwise program development in Π-institutions. Technical report, Imperial College, 1990.

[30] Mila Majster. Limits of the algebraic specification of abstract data types. *SIGPLAN Notices*, pages 37–42, October 1977.

[31] José Meseguer. General logics. In H.-D. Ebbinghaus et al., editors, *Proceedings, Logic Colloquium, 1987.* North-Holland, 1989.

[32] José Meseguer and Joseph Goguen. Initiality, induction and computability. In Maurice Nivat and John Reynolds, editors, *Algebraic Methods in Semantics*, pages 459–541. Cambridge, 1985.

[33] David Parnas. Information distribution aspects of design methodology. *Information Processing '72*, 71:339–344, 1972. Proceedings of 1972 IFIP Congress.

[34] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association for Computing Machinery*, 15:1053–1058, 1972.

[35] David Parnas. A technique for software module specification. *Communications of the Association for Computing Machinery*, 15:330–336, 1972.

[36] Pieter Hendrik Rodenburg. Interpolation in conditional equational logic, 1989. Preprint from Programming Research Group at the University of Amsterdam.

[37] Pieter Hendrik Rodenburg. A simple algebraic proof of the equational interpolation theorem. *Algebra Universalis*, 28:48–51, 1991.

[38] Donald Sannella. *Semantics, Implementation and Pragmatics of Clear, a Program Specification Language*. PhD thesis, University of Edinburgh, Computer Science Department, 1982. Technical report CST-17-82.

[39] Donald Sannella and Andrzej Tarlecki. Extended ML: an institution independent framework for formal program development. In *Proceedings, Summer Workshop on Category Theory and Computer Programming*, pages 364–389. Springer, 1986. Lecture Notes in Computer Science, Volume 240.

[40] Donald Sannella and Andrzej Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Science*, 34:150–178, 1987. Earlier version in *Proceedings, Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science, Volume 185, Springer, 1985.

[41] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988. Earlier version in *Proceedings, International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science, Volume 173, Springer, 1985.

[42] Andrzej Tarlecki. Free constructions in algebraic institutions. In *Proceedings, International Symposium on Mathematical Foundations of Computer Science*, pages 526–534. Springer, 1984. Lecture Notes in Computer Science, Volume 176; extended version, University of Edinburgh Computer Science Department Report CSR-149-83, and revised version 'On the existence of Free Models in Abstract Algebraic Institutions', September 1984.

[43] Andrzej Tarlecki. Bits and pieces of the theory of institutions. In *Proceedings, Summer Workshop on Category Theory and Computer Programming*, pages 334–360. Springer, 1986. Lecture Notes in Computer Science, Volume 240.

[44] Andrzej Tarlecki. On the existence of free models in abstract algebraic institutions. *Theoretical Computer Science*, pages 269–304, 1986. Preliminary version, University of Edinburgh, Computer Science Department Technical Report CSR-165-84, 1984.

[45] Andrzej Tarlecki. Quasi-varieties in abstract algebraic institutions. *Journal of Computer and System Sciences*, 33(3):333–360, 1986. Original version, University of Edinburgh Technical Report CSR-173-84.

[46] Władysław Turski and Thomas Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.

[47] Paulo Veloso and Sheila Veloso. Some remarks on conservative extensions: A socratic dialogue. *Bulletin of the European Association for Theoretical Computer Science*, 43:189–198, February 1991.

Contents

A Complete Proof Synthesis Method for the Cube of Type Systems

Gilles Dowek

INRIA §¶

Preliminary Version, June 1991

Abstract

We present a complete proof synthesis method for the eight type systems of Barendregt's cube extended with η-conversion. Because these systems verify the proofs-as-objects paradigm, the proof synthesis method is a one level process that merges unification and resolution. A generalization of this method to the Calculus of Constructions with universes is presented. We also present a variant of this method, which is incomplete but much more efficient. At last we study the relation between this algorithm and unification.

Introduction

The Calculus of Constructions is an extension of Church higher order logic in which the terms belong to a λ-calculus with dependent types, polymorphism and type constructors. Because of the richness of the terms structure, proofs can be represented as terms through Heyting semantics and Curry-Howard isomorphism. So a proof of a proposition P is merely a term of type P.

A proof synthesis method for Church higher order logic is given in [15]. In this paper we generalize this method to find terms of a given type in the systems of Barendregt's cube and the Calculus of Constructions with universes extended with η-conversion. In some sense, because these type systems are more powerful than Church higher order logic, the proof synthesis problem is more complicated. But we claim that because the proofs-as-objects paradigm simplifies the formalism, it also simplifies the expression of proof synthesis algorithms. In particular resolution and unification can be merged in a uniform algorithm.

We first present a complete method, then we discuss some efficiency improvements, we present a variant of this method which is incomplete but much more efficient, at last we study the relations between this method and unification.

In this paper we need to use some basic properties of type systems that are not yet fully demonstrated and which are out of the scope of this paper: we conjecture that for the eight systems of Barendregt's cube and the Calculus of Constructions with universes, $\beta\eta$-reduction is

§B.P. 105, 78153 Le Chesnay CEDEX, France. dowek@margaux.inria.fr

¶This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks".

Strongly Normalizable and Church-Rosser. Strong Normalization and Church-Rosser property of β-reduction in all these systems has been proved in [2] [4]. Strong Normalization and Church-Rosser property of $\beta\eta$-reduction in system F has been proved in [10]. Church-Rosser property of $\beta\eta$-reduction in $\lambda\Pi$-calculus has been proved in [25].

1 Outline of the Method

1.1 Resolution and Unification

In first order logic or higher order logic, we have two syntactical categories, terms and proofs. Terms are trees (in first order logic) or simply typed λ-terms (in higher order logic) and proof are build with natural deduction rules. These rule combine some proofs and terms to form new proofs. For instance the rule $\rightarrow$-elim:

$$\frac{A \rightarrow B \quad A}{B}$$

combines two proofs (one of $A \rightarrow B$ and one of A) to get a proof of B. The rule $\forall$-elim:

$$\frac{\forall x : A\ B \quad t : A}{B[x \leftarrow t]}$$

combines a proof (of $\forall x : A\ B$) and a term (of type A) to get a proof of $B[x \leftarrow t]$. So proofs are heterogeneous trees. For instance, in the proof:

$$\frac{\dfrac{\forall f : T \rightarrow T\ (P\ f) \rightarrow (Q\ f) \quad \dfrac{[x : T]}{\lambda x : T.x : T \rightarrow T}}{(P\ \lambda x : T.x) \rightarrow (Q\ \lambda x : T.x)} \quad (P\ \lambda x : T.x)}{(Q\ \lambda x : T.x)}$$

the subtree:

$$\frac{[x : T]}{\lambda x : T.x : T \rightarrow T}$$

is a term.

In first order logic and higher order logic, the *resolution* algorithm searches for proof-trees. When during the search of a proof-tree, a term-tree is needed, the *unification* algorithm is called. In type systems, term-trees and proof-trees belong to a single syntactical category, so our algorithm will be a one level process merging resolution and unification.

We shortly present now the unification and resolution algorithms for higher order logic [15] [16] [17], our goal is to show that a single idea on term enumeration underlies both algorithm.

1.2 Higher Order Unification

Higher order unification is based on an algorithm for enumerating all the normal η-long terms of a given type in the simply typed λ-calculus.

Let $T_1 \rightarrow ... \rightarrow T_n \rightarrow T$ (T atomic) be a type. All normal η-long terms of this type begin by n abstractions:

$$t = [x_1 : T_1]...[x_n : T_n]t'$$

The term t' must have type T so it is an atomic term. A variable $w : U_1 \to ... \to U_p \to U$ of the context or which is an x_i can be the head of this term, only if $U = T$. Then this variable must be applied p times in order to get a term of the good type. So:

$$t = [x_1 : T_1]...[x_n : T_n](w\ c_1\ ...\ c_p)$$

To mark the dependency of the c_i on the x_j we write:

$$t = [x_1 : T_1]...[x_n : T_n](w\ (d_1\ x_1\ ...\ x_n)\ ...\ (d_p\ x_1\ ...\ x_n))$$

To find the d_i we will use recursively the algorithm. So we first take:

$$t = [x_1 : T_1]...[x_n : T_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

then we use recursively our algorithm to instanciate the $h_i : T_1 \to ... \to T_n \to U_i$.

In unification, flexible-rigid equations are solved by instanciating the head of the flexible term using this method, the choice of the variable w is very restricted by the rigid term. Rigid-rigid equations are simplified. And flexible-flexible equations are solved in a trivial way.

So the method to enumerate terms underlying higher order unification can be summaried:

- try all the possible head variables of the normal form of the term,
- generate a typing constraint to enforce well-typedness of the term,
- generate new variables for the rest of the term,
- use recursively this method to fill these variables.

1.3 Higher Order Resolution

Let us present now the resolution method. We just present an incomplete restriction of this method to Horn clauses i.e. propositions on the form $P_1 \to ... \to P_n \to Q$, where $P_1, ..., P_n, Q$ are atomic propositions.

When the goal B is atomic we unify it with Q the head of an hypothesis $P_1 \to ... \to P_n \to Q$, we generate subgoals $\sigma P_1, ..., \sigma P_n$ and recursively look for proofs of these propositions.

When the goal is also a Horn clause $A_1 \to ... \to A_n \to B$, then the clausal form of the negation of this goal will introduce A_1, ..., A_n as hypothesis and the atomic goal B.

This restriction is called the *introduction-resolution* algorithm. It can be presented with two rules: the resolution rule between an atomic goal and the head of an hypothesis is just called the *resolution* rule and the introduction of the hypothesis A when we want to prove $A \to B$ is called the *introduction* rule. It is incomplete. In [15] another rule (the *splitting* rule) is added to make it complete.

Remark that this method can also be used for *hereditary Horn clauses* i.e. propositions on the form $P_1 \to ... \to P_n \to Q$, where Q is an atomic proposition and $P_1, ..., P_n$ are hereditary Horn clauses.

1.4 Introduction-Resolution in Type Systems

In type systems, all propositions can be considered as hereditary Horn clauses where the arrow is generalized to dependent product, so we can apply the same method.

Furthermore, in type systems, a proof of a proposition P is a term t that has type P, a proof-synthesis method has to exhibit a proof of the proposition.

The introduction-resolution algorithm can be presented that way:

• *Introduction*: To find a proof of a proposition $(x : A)B$ in a context Γ, find a proof of B in $\Gamma, x : A$. When we have a proof b of B, the term $[x : A]b$ is a proof of $(x : A)B$.

• *Resolution*: To find a proof of A in a context Γ where there is a proposition:

$$f : (x_1 : P_1)...(x_n : P_n)Q$$

unify A and Q (consider $x_1, ..., x_n$ as existential variables in Q) and then if x_i has not been instanciated during the unification process, then find a proof of σP_i. For each i, we get (by unification or as a proof of a subgoal) a term c_i of type $P_i[x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}]$. The term $(f\ c_1\ ...\ c_n)$ is a proof of A.

This method is fully described in [14]. Let us give an example. We have an hypothesis:

$$f : (x : T)(y : T)(z : T)(u : (R\ x\ y))(v : (R\ y\ z))(R\ x\ z)$$

and we want to prove $(R\ a\ c)$. We unify $(R\ x\ z)$ and $(R\ a\ c)$ this gives the substitution $x \leftarrow a, z \leftarrow c$. Then we have to find terms $b : T$, $t : (R\ a\ b)$ and $t' : (R\ b\ c)$. The term $(f\ a\ b\ c\ t\ t')$ is a proof of $(R\ a\ c)$.

This method is in general incomplete. Another drawback is that no general unification algorithm is known for type systems. Algorithms are known only for the simply typed λ-calculus [16] [17] and $\lambda\Pi$-calculus [7] [8] [23].

An alternative presentation of this method is the following:

• *Introduction*: we give the proof $[x : A]h$ for the proposition and the variable h is a subgoal to be proved.

• *Resolution*: we give the proof $(f\ h_1\ ...\ h_p)$ for the proposition together with an equation $Q[x_1 \leftarrow h_1, ..., x_p \leftarrow h_p] = A$ (in our example $(R\ h_1\ h_3) = (R\ a\ b)$) to force the term $(f\ h_1\ ...\ h_p)$ to have type A. We solve this unification problem which fills some of the variables h_i (here h_1 and h_3) and then the other variables (here h_2, h_4 and h_5) are subgoals to be proved.

When to prove a proposition we first apply the introduction rule n times then the resolution rule, we have given the proof $[x_1 : A_1]...[x_n : A_n](f\ h_1\ ...\ h_p)$ i.e. we have given f for the head variable of proof. Remark that the dependence of h_i to the x_j is implicit.

So the method to enumerate terms underlying the introduction-resolution method is:

• try all the possible head variables of the normal form of the term,

• generate a typing constraint to enforce well-typedness of the term,

• generate new variables for the rest of the term,

• use recursively this method to fill these variables.

So this method is the same as the method underlying higher order unification.

1.5 A Method to Enumerate the Terms of a Given Type

Now we will use the same idea to construct a complete proof synthesis method for type systems.

To enumerate the terms t that can be substituted to a variable x of type T we imagine the normal form of t:

$$t = [x_1 : P_1]...[x_n : P_n](w\ c_1\ ...\ c_p)$$

or:

$$t = [x_1 : P_1]...[x_n : P_n](z : A)B$$

and we perform all the elementary substitutions:

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n))$$

$$x \leftarrow [x_1 : P_1]...[x_n : P_n](z : (h_1\ x_1\ ...\ x_n))(h_2\ x_1\ ...\ x_n\ z)$$

Then we enumerate all the terms that can be substituted to the variables $h_1, ..., h_p$.

Because we consider terms in η-long form, the number of abstractions n is the number of products in the type T and the types $P_1, ..., P_n$ are the types of the variables bound in these products.

For completeness we have to consider

- all the possible w,
- all the possible p and types for $h_1, ..., h_p$.

When we perform such a substitution, we have to make sure to get a term of type T. This well-typedness constraint is an equation.

In the introduction-resolution algorithm, the solution of this constraint is called the unification step of resolution. In unification in simply typed λ-calculus, types are always ground, so the constraint relates ground terms an we only have to check that they are equal, this is what we do when we select head variables. In unification in $\lambda\Pi$-calculus [7] [8] [23] this constraint is added to the set of equations, it is the *accounting* equation.

Here we will keep this equation in the context. Equations are constraints that the future substitutions must verify. Since we do not have a special algorithm to solve these equations but use the standard method to fill the variables and use the equations as constraints that the substitutions must verify, this method is a one level process merging resolution and unification.

1.6 Number of Applications

In the introduction-resolution algorithm when we want to prove an atomic proposition T and we resolve it with a symbol $w : (y_1 : Q_1)...(y_q : Q_q)Q$ (Q atomic) we only consider proofs of the form $(w\ h_1\ ...\ h_q)$ and the constraint is that the type of this term must be equal to T i.e. $Q[y_1 \leftarrow h_1, ..., y_q \leftarrow h_q] = T$.

But in polymorphic type systems the term $(w\ c_1\ ...\ c_q)$ can still have a type which is a product and we can go on applying variables. This is why the introduction-resolution method is incomplete and why a splitting rule is needed in [15].

Here we will consider an infinite number of possibilities for the number of applications and an infinitely branching search tree, we will use interleaving to enumerate its nodes. In an incomplete but more efficient method we will consider a restriction which is more or less similar to the introduction-resolution method.

1.7 Variables in the Proposition to be Proved

In some type systems, the variables h_i may have occurrences in the type of h_j for $j > i$. We could first instanciate the variable h_i and then instanciate the variable h_j when it has a ground type, but it is well known that it is more efficient to solve first the variable h_j. For instance when we want to enumerate the terms of the type $T = \exists x : Nat\ (P\ x)$ we must not enumerate all the terms n of type Nat and for each n enumerate all the terms of type $(P\ n)$, but enumerate first the terms of type $(P\ x)$.

Thus in polymorphic type systems when we have a variable in the proposition to be proved, it is not always possible to know the number and the types of abstractions: for instance if the proposition to be proved is $T = (x_1 : P_1)...(x_n : P_n)P$ (P atomic) and the head of P is a variable then a substitution may increase the number of products in T and therefore the number of abstractions in t. So we will have to delay the instanciation of $x : T$ until we have instanciated T in a term $(x_1 : P_1)...(x_n : P_n)P$ where the head of P is not an existential variable.

2 Pure Type Systems extended with η-conversion

2.1 Definition

A Pure type system extended with η-conversion [1] (*type system* for short) is a λ-calculus defined by a set S of sorts, a set Ax of pairs of sorts and a set R of triple of sorts.

Definition: Functional Type System

A type system is said to be *functional* if:

$$< s, s' > \in Ax \text{ and } < s, s'' > \in Ax \text{ implies } s' = s''$$

$$< s, s', s'' > \in R \text{ and } < s, s', s''' > \in R \text{ implies } s'' = s'''$$

In this paper we consider systems such that $S = \{Prop, Type\}$, $Ax = \{< Prop, Type >\}$ and if $< s, s', s'' > \in R$ then $s' = s''$ and $< Prop, Prop, Prop > \in R$. These systems are the eight systems of Barendregt's cube. All these systems are functional. Examples are the simply typed λ-calculus, $\lambda\Pi$-calculus [12], system F, Fω [11] and the Calculus of Constructions without universes [2] [5].

Definition: Syntax

$$T \ ::= \ s \mid x \mid (T\ T) \mid [x : T]T \mid (x : T)T$$

In this paper we ignore variable renaming problems. A rigorous presentation would use de Bruijn indices. The terms s are sorts, the terms x are called variables, the terms $(T\ T')$ applications, the terms $[x : T]T'$ λ-abstractions and the terms $(x : T)T'$ products. The notation $T \to T'$ is used for $(x : T)T'$ when x has no free occurrence in T'.

Let t and t' be terms and x a variable. We write $t[x \leftarrow t']$ for the term obtained by substituting t' for x in t. We write $t \equiv t'$ when t and t' are $\beta\eta$-equivalent.

Definition: Context

A *context* is a list of pairs $< x, T >$ (written $x : T$) where x is a variable and T a term. The term T is called the *type* of x in Γ.

We write $[x_1 : T_1; ...; x_n : T_n]$ the context with elements $x_1 : T_1, ..., x_n : T_n$ and $\Gamma_1\Gamma_2$ for the concatenation of the contexts Γ_1 and Γ_2.

Definition: Typing Rules

We define inductively two judgements: Γ *is well-formed* and t *has type* T *in* Γ ($\Gamma \vdash t : T$) where Γ is a context and t and T are terms.

$$\overline{[]\ \text{well-formed}}$$

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \text{well-formed}}\ s \in S$$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash s : s'} < s, s' > \in Ax$$

$$\frac{\Gamma \text{ well-formed} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma[x : T] \vdash T' : s'}{\Gamma \vdash (x : T)T' : s''} < s, s', s'' > \in R$$

$$\frac{\Gamma \vdash (x : T)T' : s \quad \Gamma[x : T] \vdash t : T'}{\Gamma \vdash [x : T]t : (x : T)T'} s \in S$$

$$\frac{\Gamma \vdash t : (x : T)T' \quad \Gamma \vdash t' : T}{\Gamma \vdash (t\ t') : T'[x \leftarrow t']}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'} s \in S$$

Definition: Well-typed Term

A term t is said to be *well-typed* in a context Γ if there exists a term T such that $\Gamma \vdash t : T$.

Proposition: In a functional type system, a term t well-typed in a context Γ has a unique type modulo $\beta\eta$-equivalence.
Proof: By induction on the length of the typing derivation of t.

Definition: Type

A term T is said to be a *type* in the context Γ if there exists a sort s such that $\Gamma \vdash T : s$.

Proposition: If $\Gamma \vdash t : T$ then T is a type or a sort.
Proof: By induction on the length of typing derivations.

Proposition: We assume the $\beta\eta$-reduction relation to be strongly normalizable and Church-Rosser. Thus each term has a unique $\beta\eta$-normal form. Two terms are equivalent if they have the same $\beta\eta$-normal form.

Definition: Atomic Term

A term t is said to be *atomic* if it has the form $(u\ c_1\ ...\ c_n)$ where u is a variable or a sort. The symbol u is called the *head* of the term t.

Proposition: Let t be a normal well-typed term, t is an abstraction, a product or an atomic term.
Proof: If the term t is neither an abstraction nor a product then it can be written in a unique way:

$$t = (u\ c_1\ ...\ c_n)$$

where u is not an application.

The term u is not a product (if $n \neq 0$ because a product is of type s for some sort s and therefore cannot be applied and if $n = 0$ because t is not a product). It is not an abstraction (if

$n \neq 0$ because t is in normal form and if $n = 0$ because t is not an abstraction). It is therefore a variable or a sort.

Proposition: Let T be a well-typed normal type, the term T can be written in a unique way $T = (x_1 : P_1)...(x_n : P_n)P$ with P atomic.
Proof: By induction over the structure of T.

Definition: η-long form

Let Γ be a context and t be a $\beta\eta$-normal term well-typed in Γ and T the $\beta\eta$-normal form of its type. The *η-long form* of the term t is defined as:

- If $t = [x : U]u$ then let U' be the η-long form of U in Γ and u' be the η-long form of u in $\Gamma[x : U]$, we let the η-long form of t be $[x : U']u'$.
- If $t = (x : U)V$ then let U' be the η-long form of U in Γ and V' be the η-long form of V in $\Gamma[x : U]$, we let the η-long form of t be $(x : U')V'$.
- If $t = (w\ c_1\ ...\ c_p)$ we let $T = (x_1 : P_1)...(x_n : P_n)P$ (P atomic). We let c'_i be the η-long form of c_i in Γ, P'_i be the η-long form of P_i in $\Gamma[x_1 : P_1; ...; x_{i-1} : P_{i-1}]$ and x'_i the η-long form of x_i in $\Gamma[x_1 : P_1; ...; x_i : P_i]$. We let the η-long form of t be $[x_1 : P'_1]...[x_n : P'_n](w\ c'_1\ ...\ c'_p\ x'_1\ ...\ x'_n)$.

The well-foundedness of this definition is proved in [6].

Obviously the η-long form of a $\beta\eta$-normal term is β-normal, this motivates the definition:

Definition: β-normal η-long form

Let t be a term well-typed in a context Γ, the *β-normal η-long form* of t is the η-long form of its $\beta\eta$-normal form.

In the following all the terms are supposed to be on β-normal η-long form.

Definition: Subterm

We consider well-typed normal terms labeled with the contexts in which they are well-typed: t_Γ. Let t_Γ such a term, we define by induction over the structure of t_Γ the set $Sub(t_\Gamma)$ of *strict subterms* of t_Γ:

- if t_Γ is a sort or a variable then $Sub(t_\Gamma) = \{\}$,
- if t_Γ is an application $t = (u\ v)$ then $Sub(t_\Gamma) = \{u_\Gamma, v_\Gamma\} \cup Sub(u_\Gamma) \cup Sub(v_\Gamma)$,
- if t_Γ is an abstraction $t = [x : P]u$ then $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$,
- if t_Γ is a product $t = (x : P)u$ then $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$.

Definition: The Relation $<$

Let $<$ be the smallest transitive relation defined on normal terms such that:

- if t_Γ is a strict subterm of t'_Δ then $t_\Gamma < t'_\Delta$,
- if T_Γ is the normal (i.e. β-normal η-long) form of the type of t_Γ in Γ then $T_\Gamma < t_\Gamma$.

As proved in [6], this relation is well-founded, which means that a function that recurses both on a strict subterm and on the type of its argument is total.

2.2 The Meta Type System

Let $\mathcal{T}$ be a type system. In order to express the proof synthesis method, we have first to be allowed to declare a variable that stands for any well-typed term of the Calculus. For instance a variable that stands for *Prop* i.e. a variable of type *Type* and this is not possible in $\mathcal{T}$ because *Type* is ill-typed.

We want also to be allowed to express a term $t : T'$ which is well-typed in $\Gamma[x : T]$ as $t = (f\ x)$ with f well-formed in Γ. We want actually to be allowed to define $f = [x : T]t$ whatever the terms T and t may be. This cannot be done in the general case in $\mathcal{T}$, because the type $(x : T)T'$ may be ill-typed. So we are going to embed our type system $\mathcal{T}$ in another type system: the *Meta* type system.

Definition: Meta

$$Meta =< S', Ax', R' >$$

$$S' = \{Prop, Type, Extern\}$$

$$Ax' = \{< Prop, Type >, < Type, Extern >\}$$

$$R' = \{< Prop, Prop, Prop >, < Prop, Type, Type >, < Type, Prop, Prop >,$$

$$< Type, Type, Type >, < Prop, Extern, Extern >, < Type, Extern, Extern >\}$$

Proposition: Because all terms typable in the Meta type system are typable in the Calculus of Constructions with universes [3], we assume that in this type system the reduction relation is Strongly Normalizable and Church-Rosser. So we can define for any term t another term t' which is its β-normal η-long form, if two terms are equivalent then they have the same β-normal η-long form.

Proposition: Because the Meta type system is functional, if a term t is well-typed in a context Γ then it has a unique type modulo $\beta\eta$-equivalence.

3 Constrained Quantified Contexts

3.1 Constrained Quantified Contexts

Definition: Constrained Quantified Contexts

A *quantified 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 *constraint* is a pair of terms $< a, b >$ (written $a = b$). A *constrained quantified context* is a list $\Gamma = [e_1; ...; e_n]$ such that e_i is either a quantified declaration or a constraint. These constrained quantified contexts are generalization of Miller's *mixed prefixes* [21] which are lists of quantified declarations.

Non constrained, non quantified contexts are identified with constrained quantified contexts with no constraints and only universal variables.

Definition: Equivalence Modulo Constraints

Let Γ be a constrained quantified context, we define the relation between terms $\equiv_\Gamma$ as the smallest relation that verifies:

- if $t \equiv t'$ then $t \equiv_\Gamma t'$,

- if $(a = b) \in \Gamma$ then $a \equiv_\Gamma b$,
- if $t \equiv_\Gamma t'$ then $t' \equiv_\Gamma t$,
- if $t \equiv_\Gamma t'$ and $t' \equiv_\Gamma t''$ then $t \equiv_\Gamma t''$.

Definition: Typing Rules

First we modify the rules to deal with the new syntax:

The rule:

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T] \text{ well-formed}}\; s \in S$$

is replaced by:

$$\frac{\Gamma \vdash T : s}{\Gamma[Qx : T] \text{ well-formed}}\; s \in S$$

and we add the rule:

$$\frac{\Gamma \vdash a : T \quad \Gamma \vdash b : T}{\Gamma[a = b] \text{ well-formed}}$$

Then we extend the system by replacing the rule:

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv T'}{\Gamma \vdash t : T'}\; s \in S$$

by:

$$\frac{\Gamma \vdash T : s \quad \Gamma \vdash T' : s \quad \Gamma \vdash t : T \quad T \equiv_\Gamma T'}{\Gamma \vdash t : T'}\; s \in S$$

This defines two new judgements: Γ is *well-formed using the constraints* and t *has type* T *in* Γ *using the constraints.*

Remark: A term may be well-typed in Γ using the constraints and still be not normalizable.

Definition: Well-typed Without Using the Constraints

Let Γ be a context and t and T be two terms. The term t is said to be *of type* T *in* Γ *without using the constraints* if there exists Δ subcontext of Γ (i.e. obtained by removing some items of Γ) such that Δ has no constraints, is a well-formed context and $\Delta \vdash t : T$.

Proposition: If a term is well-typed in a context without using the constraints then it is strongly normalizable.

Definition: Normal Form of a Context

Let Γ be a context, the *normal form* of Γ is obtained by putting all the types of variables that are well typed without the constraints and all the constraints which terms are well-typed without the constraints in β-normal η-long form.

Definition: Ground Term

A term t is said to be *ground* in Γ if it has no occurrence of an existential variable.

Definition: Rigid and Flexible Terms

A normal term well-typed without the constraints which is an abstraction, a product or an atomic term $(w\ c_1\ ...\ c_n)$ with w universal variable or sort is said to be *rigid*. A normal term well-typed without the constraints which is atomic $(w\ c_1\ ...\ c_n)$ with w existential variable is said to be *flexible*.

Proposition: If Γ is a well-formed context then the context Γ' obtained by forgetting all constraints relating identical terms is well-formed, and $\Gamma \vdash t : T$ is equivalent to $\Gamma' \vdash t : T$.
Proof: By induction on the length of Γ and the length of the derivation of $\Gamma \vdash t : T$.

Definition: Equivalent Contexts

Two contexts Γ and Γ' (well-formed or not) are said to be *equivalent* if the context obtained by removing all the constraints relating identical terms are identical.

In the following we will identify equivalent contexts.

Definition: Success and Failure Contexts

A normal context Γ is said to be *a success context* if it has only universal variables, and constraints relating identical terms. It is said to be *a failure context* if there is a constraint relating two normal well-typed ground terms which are not identical.

Proposition: Let Γ be a normal context which is neither a success context nor a failure context. Then there exists an existential variable $x : T$ whose type T is well-typed without using the constraints and has the form $(x_1 : P_1)...(x_n : P_n)P$ with the term P atomic and rigid in the context $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.
Proof: Let us consider the leftmost item which is neither a universal variable nor a normal constraint relating equal terms (such an item exists since the context is not a success context).

If this item were a constraint $(a = b)$, let $\Gamma = \Delta[a = b]\Delta'$, a and b would be well-typed in Δ so, since there are neither existential variables nor (non trivial) constraints in Δ, they would be ground and well-typed without the constraints. These terms would be different and the context would be a failure context.

Thus this item is an existential variable $\exists x : T$ and T is well-typed without the constraints and ground.

3.2 Substitution

Definition: Existential Context

A context (well-formed or not) is said to be *existential* if it contains only declarations of existential variables and constraints but no declarations of universal variables.

Definition: Substitution

A finite set σ of triples $< x, \gamma, t >$ where x is a variable, γ is an existential context and t is the term is said to be a substitution if for all variable x there is at most one triple of the form $< x, \gamma, t >$ in σ.

Definition: Variable Bound by a Substitution

Let x be a variable and σ be a substitution. If the substitution σ contains a triple $< x, \gamma, t >$ then x is said to be *bound* by σ. The context γ is said to be *the context associated to x in σ*.

Definition: Substitution Applied to a Term

Let x be a variable and σ a substitution. If there is a triple $< x, \gamma, t >$ in σ then we let $\sigma x = t$ else we let $\sigma x = x$. This definition extends straightforwardly to terms.

Definition: Application of a Substitution to a Context

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 well-formed context $\sigma\Gamma$.

- If $\Gamma = []$ then σ is well-typed in Γ and $\sigma\Gamma = []$.
- If $\Gamma = \Delta[\forall x : T]$ then if σ is well-typed in Δ and if $\sigma\Delta[\forall x : \sigma T]$ is well-formed then σ is well-typed in Γ and $\sigma\Gamma = \sigma\Delta[\forall x : \sigma T]$. Otherwise σ is not well-typed in Γ.
- If $\Gamma = \Delta[\exists x : T]$ then if x is bound by σ then there exists a unique triple $< x, \gamma, t >$ in σ, if it is not we let $t = x$ and $\gamma = [\exists x : \sigma T]$.

  If σ is well-typed in Δ, $(\sigma\Delta)\gamma$ is well-formed and $(\sigma\Delta)\gamma \vdash t : \sigma T$ then σ is well-typed in Γ and $\sigma\Gamma = (\sigma\Delta)\gamma$. Otherwise σ is not well-typed in Γ.
- If $\Gamma = \Delta[a = b]$ then if σ is well-typed in Δ and $(\sigma\Delta)[\sigma a = \sigma b]$ is well-formed then σ is well-typed in Γ and $\sigma\Gamma = (\sigma\Delta)[\sigma a = \sigma b]$. Otherwise σ is not well-typed in Γ.

Definition: Equivalent Substitutions

Two substitutions σ and σ' are said to be *equivalent* if the variables bound by these substitutions are the same, and for any variable x bound by the substitutions, $\sigma x = \sigma' x$ and the contexts γ and γ' associated to x by σ and σ' are equivalent.

In the following we will identify equivalent substitutions.

Proposition: Let Γ be a context, x a variable and T, T' and u terms such that $T \equiv_\Gamma T'$ and the substitution $x \leftarrow u$ is well typed in Γ then $T[x \leftarrow u] \equiv_{\Gamma[x\leftarrow u]} T'[x \leftarrow u]$.
Proof: By induction on the length of the proof of $T \equiv_\Gamma T'$.

Proposition: If $\Delta[Qx : U]\Delta' \vdash t : T$ and $\Delta \vdash u : U$ then the context $\Delta(\Delta'[x \leftarrow u])$ is well-formed and $\Delta(\Delta'[x \leftarrow u]) \vdash t[x \leftarrow u] : T[x \leftarrow u]$.
Proof: By induction on the length of the derivation of $\Delta[Qx : U]\Delta' \vdash t : T$.

Proposition: Let Γ be a context, σ a substitution well-typed in Γ, t and T two terms such that $\Gamma \vdash t : T$. We have $\sigma\Gamma \vdash \sigma t : \sigma T$.
Proof: By induction on the number of variables of Γ bound by σ. If no variable of Γ is bound by σ the result is obvious.
Otherwise let $\exists x : P$ be the rightmost existential variable of Γ bound by σ.
We write $\Gamma = \Delta[\exists x : P][e_1; ...; e_p]$.
Let $\tau = \sigma - \{< x, \sigma x >\}$. For all term t we have $\sigma t = (\tau t)[x \leftarrow \sigma x]$. By induction hypothesis, for every context Ξ such that the substitution τ is well typed in Ξ and all terms u and U such that $\Xi \vdash u : U$ we have $\tau\Xi \vdash \tau u : \tau U$.
By a simple induction on the length of Δ we prove that σ and τ are well-typed in Δ and $\tau\Delta = \sigma\Delta$.
Then by a simple induction on p and using the property of τ we prove that τ is well-typed in Γ and:

$$\tau\Gamma = (\tau\Delta)[\exists x : \tau P][\tau e_1; ...; \tau e_p] = (\sigma\Delta)[\exists x : \tau P][\tau e_1; ...; \tau e_p]$$

Using the property of τ we have:

$$\tau\Gamma \vdash \tau t : \tau T$$

i.e.:

$$(\sigma\Delta)[\exists x : \tau P][\tau e_1; ...; \tau e_p] \vdash \tau t : \tau T$$

Let γ be the existential context associated to x by σ. Since σ is well-typed in Γ we have:

$$(\sigma\Delta)\gamma \text{ well-formed}$$

Since it is always possible to add useless declarations and constraints in a context, if the context is still well-formed:

$$(\sigma\Delta)\gamma[\exists x : \tau P][\tau e_1; ...; \tau e_p] \vdash \tau t : \tau T$$

So using the previous proposition:

$$(\sigma\Delta)\gamma[(\tau e_1)[x \leftarrow \sigma x]; ...; (\tau e_p)[x \leftarrow \sigma x]] \vdash (\tau t)[x \leftarrow \sigma x] : (\tau T)[x \leftarrow \sigma x]$$

i.e.:

$$(\sigma\Delta)\gamma[\sigma e_1; ...; \sigma e_p] \vdash \sigma t : \sigma T$$

i.e.:

$$\sigma\Gamma \vdash \sigma t : \sigma T$$

Definition: Composition of Substitutions

Let Γ be a context and σ and τ two substitutions. We define the substitution $\tau \circ \sigma$ as:

$(\tau\circ\sigma) = \{< x, \tau\gamma, \tau t > \ \mid\ < x, \gamma, t > \in \sigma\} \cup \{< x, \gamma, t > \ \mid\ < x, \gamma, t > \in \tau$ and x not bound by $\sigma\}$

where $\tau\gamma$ is defined inductively by:

- if $\gamma = []$ then $\tau\gamma = []$,
- if $\gamma = \gamma'[\exists y : U]$ and the variable y is bound by τ then $\tau\gamma = (\tau\gamma')\gamma''$ with γ'' is the context associated to y by τ,
- if $\gamma = \gamma'[\exists y : U]$ and the variable y is not bound by τ then $\tau\gamma = (\tau\gamma')[\exists y : \tau U]$,
- if $\gamma = \gamma'[a = b]$ then $\tau\gamma = (\tau\gamma')[\tau a = \tau b]$.

Proposition: Let Γ be a context and σ and τ two substitutions, such that σ is well-typed in Γ and τ is well-typed in $\sigma\Gamma$ then $\tau \circ \sigma$ is well-typed in Γ and $(\tau \circ \sigma)\Gamma = \tau\sigma\Gamma$.
Proof: By induction on the length of Γ.

- If $\Gamma = \Delta[a = b]$ then since σ is well-typed in Γ, σ is well-typed in Δ, $(\sigma\Delta)[\sigma a = \sigma b]$ is well-formed and $\sigma\Gamma = (\sigma\Delta)[\sigma a = \sigma b]$.

  Since τ is well-typed in $\sigma\Gamma$, τ is well-typed in $\sigma\Delta$, $(\tau\sigma\Delta)[\tau\sigma a = \tau\sigma b]$ is well-formed and $\tau\sigma\Gamma = (\tau\sigma\Delta)[\tau\sigma a = \tau\sigma b] = (\tau\sigma\Delta)[(\tau \circ \sigma)a = (\tau \circ \sigma)b]$

  By induction hypothesis, $(\tau \circ \sigma)$ is well-typed in Δ and $(\tau \circ \sigma)\Delta = \tau\sigma\Delta$,
  thus $((\tau \circ \sigma)\Delta)[(\tau \circ \sigma)a = (\tau \circ \sigma)b]$ is well-formed.
  So $(\tau \circ \sigma)$ is well-typed in Γ and $(\tau \circ \sigma)\Gamma = ((\tau \circ \sigma)\Delta)[(\tau \circ \sigma)a = (\tau \circ \sigma)b] = \tau\sigma\Gamma$.

- If $\Gamma = \Delta[\forall x : T]$ then since σ is well-typed in Γ, σ is well-typed in Δ, $(\sigma\Delta)[\forall x : \sigma T]$ is well-formed and $\sigma\Gamma = (\sigma\Delta)[\forall x : \sigma T]$.

  Since τ is well-typed in $\sigma\Gamma$, τ is well-typed in $\sigma\Delta$, $(\tau\sigma\Delta)[\forall x : \tau\sigma T]$ is well-formed and $\tau\sigma\Gamma = (\tau\sigma\Delta)[\forall x : \tau\sigma T] = (\tau\sigma\Delta)[\forall x : (\tau \circ \sigma)T]$

  By induction hypothesis, $(\tau \circ \sigma)$ is well-typed in Δ and $(\tau \circ \sigma)\Delta = \tau\sigma\Delta$,
  thus $((\tau \circ \sigma)\Delta)[\forall x : (\tau \circ \sigma)T]$ is well-formed.
  So $(\tau \circ \sigma)$ is well-typed in Γ and $(\tau \circ \sigma)\Gamma = ((\tau \circ \sigma)\Delta)[\forall x : (\tau \circ \sigma)T] = \tau\sigma\Gamma$.

- If $\Gamma = \Delta[\exists x : T]$ then let γ be the context associated to x by σ if there is one and $\gamma = [\exists x : \sigma T]$ if there does not. We write $\gamma = [e_1; ...; e_n]$. Since σ is well-typed in Γ, σ is well-typed in Δ, $(\sigma\Delta)[e_1; ...; e_n]$ is well-formed, $\sigma\Gamma = (\sigma\Delta)[e_1; ...; e_n]$ and $\sigma\Gamma \vdash \sigma x : \sigma T$.

  If the item e_i is the declaration of an existential variable $y : U$ bound by τ, we let γ'_i be the context associated by τ to y, if it is the declaration of an existential variable $y : U$ not bound by τ, we let $\gamma'_i = [\exists y : \tau U]$ and if it is the constraint $(a = b)$ we let $\gamma'_i = [(\tau a = \tau b)]$.

  Since τ is well-typed in $\sigma\Gamma$, it is well-typed in $\sigma\Delta$, $(\tau\sigma\Delta)\gamma'_1...\gamma'_n = (\tau\sigma\Delta)(\tau\gamma)$ is well-formed and:
  $$\tau\sigma\Gamma = (\tau\sigma\Delta)\gamma'_1...\gamma'_n = (\tau\sigma\Delta)(\tau\gamma)$$

  Since $\sigma\Gamma \vdash \sigma x : \sigma T$ we have $\tau\sigma\Gamma \vdash \tau\sigma x : \tau\sigma T$.

  Then by induction hypothesis $(\tau \circ \sigma)$ is well-typed in Δ and $(\tau \circ \sigma)\Delta = \tau\sigma\Delta$. So $((\tau \circ \sigma)\Delta)(\tau\gamma)$ is well-formed, $\tau\sigma\Gamma = ((\tau \circ \sigma)\Delta)(\tau\gamma)$ and $((\tau \circ \sigma)\Delta)(\tau\gamma) \vdash (\tau \circ \sigma)x : (\tau \circ \sigma)T$.

  Thus $(\tau \circ \sigma)$ is well-typed in Γ and $(\tau \circ \sigma)\Gamma = ((\tau \circ \sigma)\Delta)(\tau\gamma) = \tau\sigma\Gamma$.

4 A Complete Method

Let Γ be a normal context well-formed in the Meta type system such that the type of universal variables is *Prop* or *Type* but not *Extern* and which is neither a success nor a failure context. We define a set of substitutions $\Sigma(\Gamma)$.

We choose an existential variable x whose type is normal and has the form $(x_1 : P_1)...(x_n : P_n)P$ with P atomic rigid in $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ (such a variable exists because the context in neither a success nor a failure context).

Notation: If t is a term, $(\vec{x} : \vec{P})t$ is an abbreviation for $(x_1 : P_1)...(x_n : P_n)t$.

- For every w which is a universal variable declared before x or an x_i:
  $$\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n] \vdash w : (y_1 : Q'_1)...(y_q : Q'_q)Q' \ (Q' \text{ atomic})$$

  We let:
  $$Q_1 = Q'_1$$
  $$Q_2 = [y_1 : Q'_1]Q'_2$$
  $$...$$
  $$Q_q = [y_1 : Q'_1]...[y_{q-1} : Q'_{q-1}]Q'_q$$
  $$Q = [y_1 : Q'_1]...[y_q : Q'_q]Q'$$

$$w : (y_1 : Q_1)(y_2 : (Q_2\ y_1))...(y_q : (Q_q\ y_1\ ...\ y_{q-1}))(Q\ y_1\ ...\ y_q)$$

For every $r \geq 0$ we consider the *substitutions with r-splitting*:

Let s be the sort type of $(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$ and s' the sort type of P, in $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$.

For all $s_1, s'_1, ..., s_i, s'_i, ..., s_r, s'_r$ such that $< s_1, s'_1, s >\in R$, ..., $< s_i, s'_i, s'_{i-1} >\in R$ and $s'_r = s'$, we let γ be the existential context:

$$\gamma = \varphi\chi_1...\chi_r\psi$$

where:

$$\varphi = [\exists h_1 : (\vec{x} : \vec{P})Q_1;$$
$$\exists h_2 : (\vec{x} : \vec{P})(Q_2\ (h_1\ x_1\ ...\ x_n));$$
$$...;$$
$$\exists h_q : (\vec{x} : \vec{P})(Q_q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q-1}\ x_1\ ...\ x_n))]$$

and:

$$\chi_1 = [\exists H_1 : (\vec{x} : \vec{P})s_1;$$
$$\exists K_1 : (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))s'_1;$$
$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z);$$
$$\exists h_{q+1} : (\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)]$$

and for all $1 < i \leq r$:

$$\chi_i = [\exists H_i : (\vec{x} : \vec{P})s_i;$$
$$\exists K_i : (\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))s'_i;$$
$$(\vec{x} : \vec{P})(K_{i-1}\ x_1\ ...\ x_n\ (h_{q+i-1}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_i\ x_1\ ...\ x_n))(K_i\ x_1\ ...\ x_n\ z);$$
$$\exists h_{q+i} : (\vec{x} : \vec{P})(H_i\ x_1\ ...\ x_n)]$$

Finally, if $r = 0$ then we take:

$$\psi = [(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P]$$

else:

$$\psi = [(\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P]$$

The set $\Sigma(\Gamma)$ contains the substitution which associates to the variable x the existential context γ and the term:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n))$$

- If P is a sort then:

  For all sort s, such that $< s, P >\in Ax$ the set $\Sigma(\Gamma)$ also contains the substitution which associates to the variable x the empty context [] and the term:

  $$[x_1 : P_1]...[x_n : P_n]s$$

  And for all sorts s, s' such that $< s, s', P >\in R$ we let:

  $$\gamma = [\exists h : (\vec{x} : \vec{P})s \ ; \ \exists k : (\vec{x} : \vec{P})(y : (h\ x_1\ ...\ x_n))s']$$

  The set $\Sigma(\Gamma)$ also contains the substitution which associates to the variable x the existential context γ and the term:

  $$[x_1 : P_1]...[x_n : P_n](y : (h\ x_1\ ...\ x_n))(k\ x_1\ ...\ x_n\ y)$$

Definition: Derivation

A *derivation* of a context Γ is a list of contexts $[\Gamma_1; ...; \Gamma_m]$ such that $\Gamma_1 = \Gamma$, Γ_m is a success context and $\Gamma_{i+1} = \sigma_i \Gamma_i$ for $\sigma_i \in \Sigma(\Gamma_i)$.

Definition: Search tree

Let Γ be a context, we build a tree, called *the search tree* of Γ. Nodes are contexts, the root is Γ and the sons of a node Δ are all the $\sigma\Delta$ for $\sigma \in \Sigma(\Delta)$.

The success contexts and failure contexts are leaves. Success nodes are in bijection with the derivations of Γ. A semi-algorithm of proof synthesis is to enumerate the nodes of the tree in order to find a success node. Since the number of sons of a node may be infinite we have in order to get an finitely branching tree, to delay the nodes with r-splitting for r generations.

Informal Explanation:

In the solutions with 0-splitting, the q first items of γ (φ) are the declarations of the new variables $h_1, ..., h_q$ used in the substitution and the last one (ψ) is a constraint expressing the well-typedness of the substitution.

For the solutions with r-splitting ($r \geq 1$) we first declare the variables $h_1, ..., h_q$ (φ) and then we need the type $(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$ of the term $(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$ to be a product $(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$. We introduce therefore (χ_1) two existential variables H_1 and K_1 and a constraint:

$$(\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

Then we can introduce the variable h_{q+1} with the type $(\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)$.

The type of $(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n))$ is then $(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$. If $r = 1$ we just need a last constraint (ψ) expressing the well-typedness of the substitution:

$$(\vec{x} : \vec{P})(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n)) = (\vec{x} : \vec{P})P$$

and in the general case we introduce in the same way 4 items by degree of splitting and then a last constraint (ψ) for the well-typedness of the substitution.

The other substitutions provide the solutions in which the variable x is bound to a term of the form $[x_1 : P_1]...[x_n : P_n]u$ where u is a sort or a product.

5 Examples

5.1 A First Order Example

Let $\Delta = [\forall T : Prop; \forall R : T \to T \to Prop; \forall Eq : T \to T \to Prop;$
$\forall Antisym : (x : T)(y : T)(R\ x\ y) \to (R\ y\ x) \to (Eq\ x\ y); \forall a : T; \forall b : T; \forall u : (R\ a\ b); \forall v : (R\ b\ a)]$
and $\Gamma = \Delta[\exists x : (Eq\ a\ b)]$.
For x, we perform the substitution with head $Antisym$ and 0-splitting:

$$x \leftarrow (Antisym\ h_1\ h_2\ h_3\ h_4)$$

We get the context:
$\Delta[\exists h_1 : T; \exists h_2 : T; \exists h_3 : (R\ h_1\ h_2); \exists h_4 : (R\ h_2\ h_1); (Eq\ h_1\ h_2) = (Eq\ a\ b)]$
For h_1, we perform the substitution with head a and 0-splitting:

$$h_1 \leftarrow a$$

For h_2, we perform the substitution with head b and 0-splitting:

$$h_2 \leftarrow b$$

We get the context $\Delta[\exists h_3 : (R\ a\ b); \exists h_4 : (R\ b\ a)]$.
(We do not write the constraints relating identical terms.)
For h_3, we perform the substitution with head u and 0-splitting:

$$h_3 \leftarrow u$$

We get the context $\Delta[\exists h_4 : (R\ b\ a)]$.
For h_4, we perform the substitution with head v and 0-splitting:

$$h_4 \leftarrow v$$

And we get the success context Δ.
Let θ be the composition of these substitutions, $\theta x = (Antisym\ a\ b\ u\ v)$.

As we will see in the following, it is very easy to see that in this example, at each step, all the substitutions but the one considered, leed to obviously hopeless contexts.

5.2 An Example with Splitting

Let:

$$\Delta = [\forall A : Prop; \forall B : Prop; \forall I : Prop \to Prop; \forall u : (P : Prop)(I\ P) \to P; \forall v : I(A \to B); \forall w : A]$$

and $\Gamma = \Delta[\exists p : B]$.
For p, we perform the substitution with head u and 1-splitting:

$$p \leftarrow (u\ h_1\ h_2\ h_3)$$

with the new variables: $h_1 : Prop$, $h_2 : (I\ h_1)$, $H : Prop$, $K : H \to Prop$ and $h_3 : H$.
We get the context:
$\Delta[\exists h_1 : Prop; \exists h_2 : (I\ h_1); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H; h_1 = (x : H)(K\ x);$

$(K\ h_3) = B]$.
For h_1, we perform a product substitution:

$$h_1 \leftarrow (x : h')(k'\ x)$$

We get the context:
$\Delta[\exists h' : Prop; \exists k' : h' \to Prop; \exists h_2 : (I\ (x : h')(k'\ x)); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H;$
$(x : H)(K\ x) = (x : h')(k'\ x); (K\ h_3) = B]$
For h_2 we perform the substitution with head v and 0-splitting:

$$h_2 \leftarrow v$$

We get the context:
$\Delta[\exists h' : Prop; \exists k' : h' \to Prop; (I\ (x : h')(k'\ x)) = (I\ (A \to B)); \exists H : Prop; \exists K : H \to Prop;$
$\exists h_3 : H; (x : H)(K\ x) = (x : h')(k'\ x); (K\ h_3) = B]$
For h' we perform the substitution with head A and 0-splitting:

$$h' \leftarrow A$$

$\Delta[\exists k' : A \to Prop; (I\ (x : A)(k'\ x)) = (I\ (A \to B)); \exists H : Prop; \exists K : H \to Prop; \exists h_3 : H;$
$(x : H)(K\ x) = (x : A)(k'\ x); (K\ h_3) = B]$
For k' we perform the substitution with head B and 0-splitting:

$$k' \leftarrow [x : A]B$$

We get the context:
$\Delta[\exists H : Prop; \exists K : H \to Prop; \exists h_3 : H; (x : H)(K\ x) = (A \to B); (K\ h_3) = B]$
For H we perform the substitution with head A and 0-splitting:

$$H \leftarrow A$$

We get the context:
$\Delta[\exists K : A \to Prop; \exists h_3 : A; (x : A)(K\ x) = (A \to B); (K\ h_3) = B]$
For K we perform the substitution with head B and 0-splitting:

$$K \leftarrow [x : A]B$$

We get the context:
$\Delta[\exists h_3 : A]$
For h_3 we perform the substitution with head w and 0-splitting:

$$h_3 \leftarrow w$$

And we get the success context Δ.
Let θ be the composition of all these substitutions, $\theta p = (u\ (A \to B)\ v\ w)$.

6 Properties

6.1 Well-typedness

Proposition: All the types of the new existential variables and constraints are well typed in the context Γ in the Meta type system.

Proof: Since $(x_1 : P_1)...(x_n : P_n)P$ is well-typed in the Meta type system, the terms P_i are well-typed in the Meta type system of type $Prop$ or $Type$ (but not $Extern$).

The term $(y_1 : Q'_1)...(y_q : Q'_q)Q'$ is either the type of a universal variable or one of the P_i, its type is therefore $Prop$ or $Type$ but not $Extern$. So the types of the Q'_i and of Q' are $Prop$ or $Type$ but not $Extern$. So the terms Q_i and Q are well-typed in the Meta type system.

The terms $(H_i\ x_1\ ...\ x_n)$ and $(h\ x_1\ ...\ x_n)$ are well-typed in the Meta type system and have the type s with $< s, s', s'' >\in R$ for some s' and s''. So s is equal to $Prop$ or $Type$, but not $Extern$.

The terms $(Q_i\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{i-1}\ x_1\ ...\ x_n))$, s_i, s'_i, $(H_i\ x_1\ ...\ x_n)$, $(K_i\ x_1\ ...\ x_n\ z)$, $(K_i\ x_1\ ...\ x_n\ (h_i\ x_1\ ...\ x_n))$, P, s, s', $(h\ x_1\ ...\ x_n)$ and $(k\ x_1\ ...\ x_n\ y)$ are well-typed in the Meta type system and have the type $Prop$, $Type$ or $Extern$.

Then the types of existential variables and the constraints are well-typed in the Meta type system.

Proposition: All the substitutions of $\Sigma(\Gamma)$ are well-typed in Γ in the Meta type system.
Proof: For the substitution with 0-splitting we have:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$$

then using the constraint ψ:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})P$$

For the substitution with r-splitting ($r \geq 1$), we prove by induction on i that in Γ, for all $i \geq 1$:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_i\ x_1\ ...\ x_n\ (h_{q+i}\ x_1\ ...\ x_n))$$

For $i = 1$ we have:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(Q\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n))$$

so using the constraint of χ_1:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

in the context $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ we have:

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_q\ x_1\ ...\ x_n)) : (z : (H_1\ x_1\ ...\ x_n))(K_1\ x_1\ ...\ x_n\ z)$$

and since:

$$h_{q+1} : (\vec{x} : \vec{P})(H_1\ x_1\ ...\ x_n)$$

we deduce:

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n)) : (K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$$

so in the context Γ:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+1}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_1\ x_1\ ...\ x_n\ (h_{q+1}\ x_1\ ...\ x_n))$$

Then if we assume this for i:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_i\ x_1\ ...\ x_n\ (h_{q+i}\ x_1\ ...\ x_n))$$

So using the constraint of χ_i:

$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(z : (H_{i+1}\ x_1\ ...\ x_n))(K_{i+1}\ x_1\ ...\ x_n\ z)$

in the context $\Gamma[\forall x_1 : P_1; ...; \forall x_n : P_n]$ we have:

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i}\ x_1\ ...\ x_n)) : (z : (H_{i+1}\ x_1\ ...\ x_n))(K_{i+1}\ x_1\ ...\ x_n\ z)$$

and since:

$$h_{q+i+1} : (\vec{x} : \vec{P})(H_{i+1}\ x_1\ ...\ x_n)$$

we deduce:

$$(w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i+1}\ x_1\ ...\ x_n)) : (K_{i+1}\ x_1\ ...\ x_n\ (h_{q+i+1}\ x_1\ ...\ x_n))$$

so in the context Γ:

$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+i+1}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_{i+1}\ x_1\ ...\ x_n\ (h_{q+i+1}\ x_1\ ...\ x_n))$

So we deduce:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})(K_r\ x_1\ ...\ x_n\ (h_{q+r}\ x_1\ ...\ x_n))$$

then using the last constraint (ψ):

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n)) : (\vec{x} : \vec{P})P$$

Then by induction on the length of Γ and using the preservation of typing judgements by application of a well-typed substitution we prove that the substitution which associates to the variable x the existential context γ and the term:

$$[x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_{q+r}\ x_1\ ...\ x_n))$$

is well-typed in Γ.

If P is a sort, the supplementary substitutions are obviously well-typed.

6.2 Soundness

Lemma: Let Γ be a context, if there exists a derivation $[\Gamma_1; ...; \Gamma_m]$ of Γ, then there exists a substitution θ such that $\theta\Gamma$ is a success context. The substitution θ is called the substitution denoted by the derivation.

Proof: Let $\theta = \sigma_{m-1} \circ ... \circ \sigma_1$, $\theta\Gamma = \Gamma_m$ is a success context.

Now we prove that the substitution θ which is well-typed in the Meta type system is also well-typed in the original system $\mathcal{T}$.

Proposition: Let Γ be a context well-formed in a type system $\mathcal{T}$ and a type T well-typed in Γ in the system $\mathcal{T}$. Let t be term such that $\Gamma \vdash t : T$ in the Meta type system such that for every subterm of T which is a product $(x : U)U'$ if we let s be the type of U, s' be the type of U' and s'' be the type of $(x : U)U'$, we have $< s, s', s'' > \in R$ (the set of rules of the system $\mathcal{T}$) and for every subterm which is a sort s we have $s = Prop$ (and not $Type$), then we have $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

Proof: By induction over the structure of t:

- If $t = [x : U]u'$ then $T = (x : U)U'$ is well-typed in the type system $\mathcal{T}$ so U' is well-typed in the context $\Gamma[\forall x : U]$ in the type system $\mathcal{T}$ and and by induction hypothesis $\Gamma[\forall x : U] \vdash u' : U'$ in the system $\mathcal{T}$. So $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

- If $t = (x : U)U'$ then by induction hypothesis U and U' are well-typed in the system $\mathcal{T}$ and since the rule $< s, s', s'' >$ is a rule of the system $\mathcal{T}$, we have $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

- If $t = (x\ c_1\ ...\ c_n)$ with x variable declared in Γ then x is well-typed in Γ in the system $\mathcal{T}$ and we prove by induction on i that the type of c_i is well-typed in the system $\mathcal{T}$ and c_i is well-typed in the system $\mathcal{T}$ and we conclude that $\Gamma \vdash t : T$ in the system $\mathcal{T}$.

- If t is a sort then by hypothesis $t = Prop$, it is well-typed in the system $\mathcal{T}$.

Proposition: Let Γ be a context well-formed in the type system $\mathcal{T}$, $[\Gamma_1; ...; \Gamma_m]$ be a derivation of Γ and θ be the substitution denoted by this derivation. The substitution θ is well-typed in Γ in the system $\mathcal{T}$.
Proof: By induction on the length of Γ.

Theorem: Soundness

Let Γ be a (non constrained, non quantified) context and P a well-typed type in Γ. Let $\Gamma_1 = \Gamma[\exists x : P]$. If there exists a derivation of Γ_1 then there exists a proof t of P in Γ in $\mathcal{T}$. The proof t is called the proof denoted by the derivation.
Proof: Let θ be a substitution denoted by the derivation of Γ_1. The substitution θ is well-typed in Γ_1 in the system $\mathcal{T}$ and $\Gamma_m = \theta\Gamma_1$ is a success context. Since the term P is ground $\theta P = P$.

Let $t = \theta x$ and γ the context associated to x by θ, $\theta\Gamma = \Gamma$, so $\theta\Gamma_1 = \Gamma_m = \Gamma\gamma \vdash t : P$. The context Γ_m is a success context and γ is an existential context so this context is a list of constraints that relate equal terms, so $\Gamma \vdash t : P$.

6.3 Completeness

Definition: Size of a Term

Let Γ be a context and t a term well-typed in Γ. Let T be the normal form of the type of t in Γ. We define by induction over $<$, the *size* of t_Γ ($|t_\Gamma|$):

- If t is a sort then $|t_\Gamma| = 1$,

- If t is a variable then $|t_\Gamma| = |T_\Gamma|$,

- If $t = (u\ v)$ then $|t_\Gamma| = |u_\Gamma| + |v_\Gamma| + |T_\Gamma|$,

- If $t = [x : U]u$ then $|t_\Gamma| = |u_{\Gamma[x:U]}|$,

- If $t = (x : U)V$ then $|t_\Gamma| = |U_\Gamma| + |V_{\Gamma[x:U]}| + |T_\Gamma|$.

Definition: Size of a Substitution

The *size* of a substitution $\theta = \{< x_i, \gamma_i, t_i >\}$ such that all the t_i are normal, is the sum of the sizes of the t_i.

Proposition: If Γ is a failure context, then for every substitution σ well-typed in Γ, $\sigma\Gamma$ is also a failure context and therefore is not a success context.
Proof: Let $a = b$ be a constraint of Γ relating two ground different terms, then $(\sigma a = \sigma b)$ is $(a = b)$ which is a constraint relating two well-typed ground different terms.

Lemma: Let Γ be a context and θ a substitution such that $\theta\Gamma$ is a success context, then there exists a derivation of Γ that denotes the substitution θ.
Proof: By induction on the size of θ.

Since $\theta\Gamma$ is a success context, all the contexts associated to existential variables of θ are lists of constraints relating ground equal terms. So we can consider that these contextes are empty. The context Γ is not a failure context because there exists a substitution θ such that $\theta\Gamma$ is a success context. If Γ is a success context then $[\Gamma]$ is a derivation of Γ.

Else let us chose an existential variable x with a normal type $(x_1 : P_1)...(x_n : P_n)P$ such that P is atomic rigid.

Let $t = \theta x$. The term t is of type $(x_1 : \theta P_1)...(x_n : \theta P_n)\theta P$. Since the term P is atomic rigid, the term θP is atomic. Then $t = [x_1 : \theta P_1]...[x_n : \theta P_n]u$ with u atomic or a product (η-long form).

- If $u = (w\ u_1\ ...\ u_p)$ then let q the number of products of the type of w and $r = p - q$. Since the type of u is atomic we have $p \geq q$, i.e. $r \geq 0$.

  We let:

  $$\sigma = \{< x, \gamma, [x_1 : P_1]...[x_n : P_n](w\ (h_1\ x_1\ ...\ x_n)\ ...\ (h_p\ x_1\ ...\ x_n)) >\}$$

  with γ defined in the algorithm.

  Then we build the terms to be substituted to the variables h_i $(1 \leq i \leq p)$ and H_i and K_i $(1 \leq i \leq r)$. We let:

  $$u_i' = [x_1 : P_1]...[x_n : P_n]u_i$$

  And for all $1 \leq i \leq r$, $(w\ u_1\ ...\ u_{q+i-1})$ has a type which is a product, let $(y : U_i)V_i$ be this product. We let:

  $$U_i' = [x_1 : P_1]...[x_n : P_n]U_i$$

  $$V_i' = [x_1 : P_1]...[x_n : P_n][y : U_i]V_i$$

  $$\theta' = \theta - \{< x, [], \theta x >\} \cup \{< h_i, [], u_i' >\} \cup \{< H_i, [], U_i' >, < K_i, [], V_i' >\}$$

  We have $\theta = \theta' \circ \sigma$.

  Let us prove that the substitution θ' is smaller than θ: $\theta x = [x_1 : \theta P_1]...[x_n : \theta P_n](w\ u_1\ ...\ u_p)$.
  Let T_i be the type of $(w\ u_1\ ...\ u_i)$.
  $|\theta x| = |(w\ u_1\ ...\ u_p)| = |T_0| + |u_1| + |T_1| + ... + |u_p| + |T_p|$
  $> |T_q| + ... + |T_{p-1}| + |u_1| + ... + |u_p| = |(y : U_1)V_1| + ... + |(y : U_r)V_r| + |u_1| + ... + |u_p|$
  $> |U_1| + ... + |U_r| + ... + |V_1| + ... + |V_r| + |u_1| + ... + |u_p|$
  $= |U_1'| + ... + |U_r'| + |V_1'| + ... + |V_r'| + |u_1'| + ... + |u_p'|$
  So $|\theta| > |\theta'|$.

- If $u = (x : U)V$ then we let:

  $$\sigma = \{< x, \gamma, [x_1 : P_1]...[x_n : P_n](y : (h\ x_1\ ...\ x_n))(k\ x_1\ ...\ x_n\ y) >\}$$

with γ defined in the algorithm.

Then we build the terms to be substituted to the variables h and k. We let:

$$U' = [x_1 : P_1]...[x_n : P_n]U$$

$$V' = [x_1 : P_1]...[x_n : P_n][y : U]V$$

$$\theta' = \theta - \{< x, [], \theta x >\} \cup \{< h, [], U' >, < k, [], V' >\}$$

We have $\theta = \theta' \circ \sigma$.

Let us prove that the substitution θ' is smaller than θ: $\theta x = [x_1 : \theta P_1]...[x_n : \theta P_n](y : U)V$
$|\theta x| = |(y : U)V| > |U| + |V| = |U'| + |V'|$
So $|\theta| > |\theta'|$.

In both cases, by induction hypothesis, there exists a derivation D of $\sigma\Gamma$ that denotes θ' and $[\Gamma]D$ is a derivation of Γ that denotes θ.

Theorem: Completeness

Let Γ be a (non constrained, non quantified) context and P a type well-typed in Γ such that there exists a term t such that $\Gamma \vdash t : P$ then there exists a derivation of $\Gamma_1 = \Gamma[\exists x : P]$ which denotes the proof t.

Proof: We let $\theta = \{< x, [], t >\}$. $\theta\Gamma_1 = \Gamma$, so there exists a derivation of Γ_1 that denotes θ.

7 Efficiency Improvements

To improve the efficiency of this method we have to recognize nodes that cannot lead to success nodes and prune the search tree.

7.1 Incremental Partial Checking of Constraints

First we must perform incremental partial checking of constraints. So we define a function of simplification, which is very similar to the SIMPL function from [16] [17]. In order to define this function we have to modify a little the syntax of constraints: a constraint is now a triple $< \delta, a, b >$ where δ is a context with universal variables only. If $\Gamma = \Delta[< \delta, a, b >]\Delta'$ then the terms a and b must be well-typed and have the same type in $\Delta\delta$.

Definition: Simplification

We consider only constraints well-typed without the constraints. While there are rigid-rigid constraints in C and the simplification has not failed we iterate the process of replacing Γ by Γ'.

Let $\delta, a = b$ be a rigid-rigid constraint of Γ. $\Gamma = \Delta[\delta, a = b]\Delta'$. We define Γ' as:

- if a and b are both abstractions $a = [x : T]a'$, $b = [x : U]b'$, since the terms a and b have the same type in $\Delta\delta$ we have $T = U$,
  we let $\Gamma' = \Delta[\delta[\forall x : T], a' = b']\Delta'$,
- if a and b are both products $a = (x : T)a'$, $b = (x : U)b'$ and T and U have the same type, we let $\Gamma' = \Delta[\delta, T = U][\delta[\forall x : T], a' = b']\Delta'$,

- if a and b are both atomic with the same head variable and they have the same number of arguments, $a = (w\ u_1\ ...\ u_p)$, $b = (w\ t_1\ ...\ t_p)$, we let $\Gamma' = \Delta[\delta, u_1 = t_1]...[\delta, u_p = t_p]\Delta'$,

- in the other cases the simplification fails.

If the simplification fails then the branch is hopeless, it can be pruned.

7.2 Using the Flexible-Rigid Constraints

When we have a flexible-rigid constraint, and we can solve the head variable of the flexible term (i.e. when the head of its type is rigid) we must solve it first, the only candidates for the head variable in an elementary substitution are the bound variables and the head of the rigid term [16] [17]. The other variables lead to unsolvable rigid-rigid constraints.

7.3 Using the Flexible-Flexible Constraints

When we have a flexible-flexible constraint and we can solve one of the head variables, we must solve it first, the constraint will not help to restrict the number of substitutions, but it may become a flexible-rigid constraint next step. Moreover solving constraints helps to unfreeze types of existential variables and constraints that are not well-typed without the constraints.

7.4 Avoiding Splitting

The term $(Q\ (h_1\ x_1\ ...\ x_n)...(h_q\ x_1\ ...\ x_n))$ is atomic, if it is rigid then all the solutions with r-splitting $r \neq 0$ lead to hopless constraints and therefore they can be avoided. In particular in $\lambda\Pi$-calculus, we never consider existential variables for types or predicates and so splitting can always be avoided.

7.5 Solving some Decidable Unification Problems

When we have a constraint well-typed without the constraints which is either a first order unification problem [24], an argument-restricted unification problem [21] [22], a second order matching problem [17] [19] [6], a second-order-argument-restricted matching problem [6], an equation $x = t$ with t well-typed in the prefix of the context Γ defined before x or an equation $(x\ c_1\ ...\ c_n) = t$ with x existential variable, $c_1, ..., c_n$ universal variables declared after x and t well-typed in the prefix of Γ truncated before x then we can solve it using always terminating algorithms and apply the substitutions obtained in this way.

In a system with such heuristics, a first-order $\forall$-elim is a one-step operation, and higher-order $\forall$-elim is a more complex operation that needs several steps and can lead to backtracking.

7.6 Normalizing some ill-typed terms

An important improvement of the method would be to recognize some normalizable ill-typed terms and then be able to use the constraints more efficiently. For instance it is possible to prove that if t is a normal term and σ a substitution $x \leftarrow [x_1 : P_1]...[x_n : P_n]u$ where u is a product or an atomic term with a head variable which is a universal variable or a sort (but not one of the x_i), then σt is normalizable.

8 Extension to The Calculus of Constructions with Universes

In an extension to the Calculus of Constructions with universes [3] [20] we do not need a Meta type system any more because all the term we consider in the algorithm in this system are well-typed in this system. But we have a new problem: when we want to instanciate $x : Prop$ by a product we have an infinite number of solutions $(y : h)(k\ y)$ with $h : Type(i)$ and $k : (y : h)Prop$. This problem can be solved by using the floating universes [18] [13].

When we have a cumulativity rule, a well-typed term does not have a unique type any more (this problem can also be solved by using the floating universes) and constraints are not equalities any more but inequalities for the cumulativity relation.

9 An Incomplete but more Efficient Method

In the second example, in order to get the proof $(u\ (A \to B)\ v\ w) : B$ we have considered the elementary substitution with 1-splitting:

$$p \leftarrow (u\ h_1\ h_2\ h_3)$$

with the three applications although the type of u begins by only two products.

Because we have to consider all these possible degree of splitting, the method is quite inefficient, we get a much more efficient one if we restrict the rule to elementary substitutions 0-splitting and a number of abstractions ranging from 0 to the number of products of the type of the existential variable we substitute. Let us call this algorithm *algorithm with weak splitting* (by opposition the previous one can be called *algorithm with strong splitting*). The search tree of the weak splitting algorithm is finitely branching.

This algorithm with weak splitting cannot synthesize a proof of B, but it can synthesize a proof of $A \to B$. Indeed with the variable $p' : A \to B$ we perform the substitution:

$$p' \leftarrow (u\ h_1\ h_2)$$

with the types $h_1 : Prop$, $h_2 : (I\ h_1)$ (remark that we have no abstraction although the type of p' begins by a product). The constraint $h_1 = A \to B$ suggests the substitution:

$$h_1 \leftarrow (A \to B)$$

then we have an existential variable $h_2 : (I\ (A \to B))$ we perform the substitution:

$$h_2 \leftarrow v$$

The synthesized proof is $(u\ (A \to B)\ v) : (A \to B)$. Remark that this term is not in η-long form, this form is $[w : A](u\ (A \to B)\ v\ w) : (A \to B)$.

In both proofs (of B and $A \to B$) the problem is to notice that the term $u : (P : Prop)(I\ P) \to P$ must be applied to the proposition $(A \to B)$. With the strong splitting algorithm this is found in both cases, but the price to pay is inefficiency, with the weak splitting this is found when the type of the existential variable is $A \to B$ but not when it is B. Roughly speaking, this means that when we must use a proposition with a quantifier on a proposition or a predicate (for instance an induction axiom) the weak splitting algorithm finds the proposition to be used only when it *can be seen* in the type of the existential variable, and not when the proof requires an *induction loading* [14] (as here from B to $A \to B$).

This weak splitting algorithm is quite similar to the introduction-resolution method [14] for some unification algorithm. Actually some proofs that cannot be synthesized by introduction-resolution can be synthesized with weak splitting. For instance in a context that contains the universal variables $u : (P : Prop)(P \to A)$ and $v : (B \to C)$ the proof $(u\ (B \to C)\ [x : B](v\ x)) : A$ is synthesized by the weak-splitting algorithm and not by the introduction-resolution method.

This method with weak splitting is incomplete but its transitive closure is complete. This means that even if it cannot be synthesized each proof in can be broken up in smaller proofs that can be synthesized.

10 Unification

10.1 Ground Unification

The problem of unification is to decide if given a context Γ (with universal and existential variables but no constraints) and two terms a and b well-typed in Γ with the same type, there exists a substitution θ well-typed in Γ, such that for all variable x bound by θ, the context associated to x by θ contains no constraints and such that $\theta a = \theta b$. In first order unification and higher order unification we do not require θ to fill all the existential variables of Γ. Moreover in higher order unification, the substitution θ may introduce new existential variables. This tolerance is due to the fact that in first order and higher order logic, types are all supposed to be inhabited, so from any unifier we can deduce a ground unifier by filling the unfilled variables with arbitrary terms. So unifiability is equivalent to ground unifiability. This is not the case any more in type systems where we have empty types. In unification in simply typed λ-calculus under a mixed prefix [21] all the substitution considered are required to fill all the existential variables and the notion of unifiability considered is ground unifiability.

A semi-algorithm for deciding ground unifiability is to apply the method developed in this paper to the context $\Gamma[a = b]$. Moreover this algorithm enumerates all the ground unifiers.

In contrast with what happens in simply typed λ-calculus, we have to solve flexible-flexible equations. Indeed these equations may have no solutions since the head variables of the terms may have empty types. Moreover it is proved in [21] that the problem of deciding the existence of solutions for flexible-flexible unification problems under a mixed prefix is undecidable in simply typed λ-calculus. This proofs generalizes easily to type systems.

10.2 Toward Open Unification

The utility of an open unification algorithm is not obvious, since in a proof-search algorithm (or more generally in an algorithm that uses unification) existential variables are introduced to be filled-in and not to remain forever.

If we still need such an algorithm we may remark that flexible-flexible equations have to be solved since, in contrast with what happens in simply typed λ-calculus, they may have no solutions.

Consider:

$$[\forall A : Prop; \forall B : Prop; \forall u : A; \exists x : (P : Prop)P]$$

$$a = (x\ (A \to B)\ u) \qquad b = (x\ B)$$

The equation $a = b$ is flexible-flexible, but there is no solution to this problem. Indeed, let us suppose there is one and consider $t = \theta x$. We have $t : (P : Prop)P$ so $t = [P : Prop]t'$ with

$t' : P$. So t' is neither an abstraction nor a product. It is an atomic term $(f\ c_1\ ...\ c_n)$. If f were the variable P we would have $n = 0$ and thus $t' = P$ which is impossible for type reasons, so $f \neq P$. Since $(t'[P \leftarrow (A \rightarrow B)]\ u) = t'[P \leftarrow B]$ we have:

$$(f\ c_1[P \leftarrow (A \rightarrow B)]\ ...\ c_n[P \leftarrow (A \rightarrow B)]\ u) = (f\ c_1[P \leftarrow B]\ ...\ c_n[P \leftarrow B])$$

So $(f\ c_1[P \leftarrow (A \rightarrow B)]) = f$ which is impossible since a variable cannot be equal to an application.

Searching for solutions of flexible-flexible equations is be rather difficult, since the head variable of an elementary substitution may now be any variable of Γ and also any new variable. We would have to design an algorithm similar to the one described in this paper but that enumerates all the proofs of a propositions, under any set of axioms.

We may think that it is possible to recognize flexible-flexible equations that have solutions and those that have not, and keep the former unsolved until the end of the process as we do in unification in simply typed λ-calculus. In this case we also have to worry about the normalization of terms well-typed using such constraints. We may either prove that such terms are still head-normalizable (as in the case of unification in $\lambda\Pi$-calculus) or perform one normalization step as an elementary operation of the algorithm.

Also some the product substitutions have to be performed when the types of the variables to be instanciated in flexible-rigid and flexible-flexible equations has an existential head variable.

In contrast with ground unifiability, open unifiability seems a quite difficult problem with small interest.

Acknowledgements

The author thanks Gérard Huet who has supervised this work and Amy Felty and Christine Paulin for many helpful comments and criticisms.

References

[1] H. Barendregt, Introduction to Generalized Type Systems, To appear in *Journal of Functional Programming.*

[2] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université Paris VII, 1985.

[3] Th. Coquand, An analysis of Girard's paradox, *Proceedings of Logic in Computer Science*, 1986, pp. 227-236.

[4] Th. Coquand, J. Gallier, A Proof of Strong Normalization For the Theory of Constructions Using a Kripke-Like Interpretation, Personal communication.

[5] T. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[6] G. Dowek, A Second Order Pattern Matching Algorithm in the Cube of Typed λ-Calculi. *Proceedings of Mathematical Foundation of Computer Science*, 1991. Rapport de Recherche, INRIA, 1991.

[7] C. M. Elliott, Higher-order Unification with Dependent Function Types, *Proceedings of the 3^rd International Conference on Rewriting Techniques and Applications*, N. Dershowitz (Ed.), Lecture Notes in Computer Science, 355, Springer-Verlag, 1989, pp. 121-136.

[8] C. M. Elliott, Extensions and Applications of Higher-order Unification, *PhD Thesis*, Carnegie Mellon University, Pittsburgh, 1990.

[9] H. Geuvers, M.J. Nederhof, A Modular Proof of Strong Normalization for the Calculus of Constructions, *Catholic University Nijmegen.*

[10] J. Gallier, On Girard's Candidats de Réductibili¡té, *Logic and Computer Science*, P. Odifreddi (Ed.), Academic Press, London, 1990, pp. 123-203.

[11] J.Y. Girard, Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur, *Thèse de Doctorat d'État*, Université de Paris VII, 1972.

[12] R. Harper, F. Honsell, G. Plotkin, A Framework for Defining Logics, *Proceedings of Logic in Computer Science*, 1987, pp. 194-204.

[13] R. Harper, R. Pollack, Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions, *CCIPL, TAPSOFT'89*, Barcelona, 1989.

[14] L. Helmink, Resolution and Type Theory, *Proceedings of the ESOP Conference*, Copenhagen, Lecture Notes in Computer Science, 432, Springer-Verlag, 1990.

[15] G. Huet, Constrained Resolution A Complete Method for Higher Order Logic, *PhD Thesis*, Case Western Reserve University, 1972.

[16] G. Huet, A Unification Algorithm for Typed λ-calculus, *Theoretical Computer Science*, 1, 1975, pp. 27-57.

[17] G. Huet, Résolution d'Équations dans les Langages d'Ordre 1,2, ..., ω, *Thèse de Doctorat d'État*, Université de Paris VII, 1976.

[18] G. Huet, Adding Type:Type to the Calculus of Constructions, Unpublished Note, 1988.

[19] G. Huet, B. Lang, Proving and Applying Program Transformations Expressed with Second Order Patterns, *Acta Informatica*, 11, 1978, pp. 31-55.

[20] Z. Luo, An Extended Calculus of Constructions, *PhD Thesis*, University of Edinburgh, 1990.

[21] D. A. Miller, Unification Under a Mixed Prefix, To appear in *Journal of Symbolic Computation.*

[22] F. Pfenning, Unification and anti-Unification in the Calculus of Constructions, To appear in *Proceedings of Logic in Computer Science*, 1991.

[23] D. Pym, Proof, Search and Computation in General Logic, *PhD thesis*, University of Edinburgh, 1990.

[24] J. A. Robinson, A Machine-Oriented Logic Based on the Resolution Principle, *Journal of the Association for Computing Machinery*, 12, 1, 1965, pp. 23-41.

[25] A. Salvesen, The Church-Rosser Theorem for LF with β/η-reduction, Manuscript, University of Edinburgh, 1989.

A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs *

Amy Felty

INRIA Rocquencourt
Domaine de Voluceau
78153 Le Chesnay Cedex, France

Abstract

In this paper, we show that an intuitionistic logic with second-order function quantification, called hh^2 here, can serve as a meta-language to directly and naturally specify both sequent calculi and natural deduction inference systems for first-order logic. For the intuitionistic subset of first-order logic, we present a set of hh^2 formulas which simultaneously specifies both kinds of inference systems and provides a direct and concise account of the correspondence between cut-free sequential proofs and normal natural deduction proofs. The logic of hh^2 can be implemented using such logic programming techniques as providing operational interpretations to the connectives and implementing unification on λ-terms. With respect to such an interpreter, our specification provides a description of how to convert a proof in one system to a proof in the other. The operation of converting a sequent proof to a natural deduction proof is functional in the sense that there is always one natural deduction proof corresponding to each sequent proof. Our specification, in fact, provides a direct implementation of the transformation in this direction.

1 Introduction

Intuitionistic logic with quantification at all function types has been proposed as a meta-language for the direct and natural specification of a wide range of logics [5, 12]. As illustrated by these papers, the second-order subset of this logic, called hh^2 here, is all that is required to suitably specify both sequent calculi and natural deduction for first-order logic. From these specifications, it is possible to obtain a description of goal-directed theorem proving and proof checking for these inference systems by providing simple operational interpretations for the connectives of hh^2.

The correspondence between cut-free sequential proofs and normal natural deduction proofs for first-order intuitionistic logic has been formalized in [16, 13]. (There, the formal relation between cut-elimination and proof normalization is also explored in detail.) In this paper, we show that it is possible to merge an hh^2 specification of a sequent system for first-order intuitionistic logic with one for natural deduction, resulting in a specification which demonstrates

*Appears in *Extensions of Logic Programming*, Springer-Verlag LNCS 475, Peter Schroeder-Heister, ed., 1991.

the correspondence between these two systems in a concise and natural manner. The operational interpretations of the connectives of hh^2 provide descriptions for simultaneous proof construction and proof checking in these two inference systems, as well as a description of how to convert a proof in one system to a proof in the other.

The transformation from sequent proofs to natural deduction proofs is "functional" since the relation between sequent proofs and natural deduction proofs is many to one, *i.e.*, there will always be exactly one normal natural deduction proof corresponding to each cut-free sequent proof. Our specification in fact provides a direct implementation of the transformation in this direction with respect to a deterministic logic programming interpreter which implements hh^2. Our specification cannot, however, serve directly as a program for the transformation in the other direction. More control information must be added in order to obtain a program that can transform a natural deduction proof to one of the possibly many sequent proofs to which it is related.

The logic of hh^2 is a sublanguage of *higher-order hereditary Harrop formulas* which serve as the foundation of the logic programming language λProlog [11]. In particular, it is the sublanguage with function quantification restricted to second-order and with no predicate quantification. Thus, an implementation of λProlog includes an interpreter for hh^2. The transformation from sequent proofs to natural deduction proofs has in fact been implemented and tested in λProlog.

In Sect. 2 we present the meta-logic hh^2 and an interpreter for it. In Sect. 3 we discuss the specification of a sequent system for first-order intuitionistic logic using this meta-logic. In Sect. 4 we will see that there are many options in specifying natural deduction. We first present a direct specification of the inference rules and then discuss alternatives, leading to a specification in which only normal proofs can be constructed and which can be merged directly with the sequent specification. Then, in Sect. 5 we demonstrate how these two specifications can be merged, and finally, in Sect. 6 we conclude.

2 The Meta-Logic and Language

The logic of hh^2 extends positive Horn clauses in essentially two ways. First, it replaces first-order terms with the more expressive simply-typed λ-terms. Second, it permits richer logical expressions in both goals and the bodies of program clauses. In particular, implication and second-order universal quantification over functions are permitted.

The types and terms of this language are essentially those of the simple theory of types [1]. We assume a fixed set of *primitive types*, which includes at least the symbol o, the type for propositions. *Function types* are constructed using the binary infix symbol $\rightarrow$; if τ and σ are types, then so is $\tau \rightarrow \sigma$. The type constructor $\rightarrow$ associates to the right. The *order* of a primitive type is 0 while the order of a function type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where τ_0 is primitive is one greater than the maximum order of $\tau_1, \ldots, \tau_n$.

For each type τ, we assume that there are denumerably many constants and variables of that type. A *signature* is a finite set of constants, such that for each constant c, the type of c has the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 0$, τ_0 is primitive, and $\tau_1, \ldots, \tau_n$ do not contain o. If τ_0 is o, then c is a *predicate constant.* In this paper, the types of constants in signatures will always be of order 2 or less.

Simply typed λ-terms are built in the usual way using constants, variables, applications, and abstractions. Equality between λ-terms is taken to mean $\beta\eta$-convertible. We shall assume that the reader is familiar with the usual notions and properties of substitution and α, β, and η conversion for the simply typed λ-calculus. See [7] for a fuller discussion of these basic properties.

The logical connectives are defined by introducing suitable constants as in [1]. The constants $\wedge$ (conjunction) and $\supset$ (implication) are both of type $o \to o \to o$, and $\forall_\tau$ (universal quantification) is of type $(\tau \to o) \to o$, for all types τ not containing o. The expression $\forall_\tau(\lambda z\ t)$ is written simply as $\forall_\tau z\ t$. An *atomic formula* is of the form $(pt_1 \dots t_n)$, where $n \geq 0$, p is a predicate constant of the type $\tau_1 \to \cdots \to \tau_n \to o$, and $t_1, \dots, t_n$ are terms of the types $\tau_1, \dots, \tau_n$, respectively. We write A to denote a syntactic variable for atomic formulas. We define two classes of *non-atomic formulas* called *goal formulas* and *definite clauses*. Let G be a syntactic variable for goal formulas and let D be a syntactic variable for definite clauses. These two classes are defined by the following mutual recursion.

$$G := A \mid G_1 \wedge G_2 \mid D \supset G \mid \forall_\tau x\ G$$

$$D := A \mid G \supset A \mid \forall_\tau x\ D$$

Here τ is a type of order 0 or 1 not containing o. We say that a formula has order n if in each of its subformulas of the form $\forall_\tau z\ t$, type τ has order strictly less than n. Thus all goal formulas and definite clauses are second-order. A *logic program* or just simply a *program* is a finite set, generally written as $\mathcal{P}$, of closed definite clauses. In definite clauses of the form $\forall_{\tau_1} x_1 \dots \forall_{\tau_n} x_n\ (G_1 \wedge \cdots \wedge G_n \supset A)$, we call A the *head* and $G_1 \wedge \cdots \wedge G_n$ the *body*. The subformulas $G_1, \dots, G_n$ are also called *subgoals*.

Based on properties shown for the full logic of higher-order hereditary Harrop formulas in [9], a sound and complete (with respect to intuitionistic logic) *non-deterministic* search procedure can be described using the following four search primitives.

AND: $G_1 \wedge G_2$ is provable from $\mathcal{P}$ if and only if both G_1 and G_2 are provable from $\mathcal{P}$.

GENERIC: $\forall_\tau x G$ is provable from $\mathcal{P}$ if and only if $[c/x]G$ is provable from $\mathcal{P}$ for some new constant c of type τ not in $\mathcal{P}$ or G.

AUGMENT: $D \supset G$ is provable from $\mathcal{P}$ if and only if G is provable from $\mathcal{P} \cup \{D\}$.

BACKCHAIN: The atomic formula A is provable from $\mathcal{P}$ if and only if there is either (1) a universal instance of a definite clause in $\mathcal{P}$ that is $\beta\eta$-convertible to A, or (2) a universal instance of the form $G \supset A$ such that G is provable from $\mathcal{P}$.

In order to implement a *deterministic* interpreter, it is necessary to make choices left unspecified in the high-level description above. We will make choices as in the λProlog language, many of which are similar to those made in Prolog. For example, logic variables are employed in the BACKCHAIN operation to create universal instances of definite clauses. As a result, second-order unification is necessary since logic variables for functions (with types of order one) can occur inside λ-terms. Also the equality of terms is not a simple syntactic check but a more complex check of $\beta\eta$-conversion. Second-order unification is not in general decidable. In λProlog, full higher-order unification is required and this issue is addressed by implementing a depth-first version of the unification search procedure described in [8]. (See [10, 9].) In this paper, the second-order unification problems that result from programs we present are all rather simple: it is easy to see, for example, that all such problems are decidable.

In the AUGMENT search operation, clauses get added to the program dynamically. Note that as a result, clauses may in fact contain logic variables. The GENERIC operation must be implemented so that the new constant c introduced for x, must not appear in the terms eventually instantiated for logic variables free in the goal or in the program when c is introduced.

Finally, we assume conjuncts are attempted in the order they are presented, and definite clauses are backchained over in the order they are listed in $\mathcal{P}$ using a depth-first search paradigm to handle failures.

In presenting example hh^2 formulas in this paper, we will adopt the syntax of the eLP [3] implementation of λProlog. Fortunately, very little of this syntax is needed here. In particular, a signature member, say f of type $a \rightarrow b \rightarrow c$, is represented as simply the line:

```
type   f        a -> b -> c.
```

Tokens with initial capital letters will denote either bound or free variables. All other tokens will denote constants. λ-abstraction is written using backslash \ as an infix operator. Universal quantification is written using the constant `pi` in conjunction with a λ-abstraction, *e.g.*, `pi X\` represents universal quantification over variable `X`. We omit type subscripts for this quantifier. They can always be inferred from context. The symbols `,` and `=>` represent $\wedge$ and $\supset$, respectively. The symbol `:-` denotes the converse of `=>` and is used to write the top-level implication in definite clauses. We omit universal quantifiers at the top level in definite clauses, and assume implicit quantification over all free variables.

Consider a set of primitive types that includes the types `a` and `a_l`. Here, `a_l` is meant to represent the type for lists of elements of type `a`. Consider a signature which includes the following declarations:

```
type    nil_a   a_l.
type    ::_a    a -> a_l -> a_l.
type    memb_a  a -> a_l -> o.
```

where `nil_a` represents the empty list of elements of type `a`, `::_a` is the cons operator for `a`-lists, and `memb_a` is a predicate which takes as arguments an element and a list of elements of type `a`. We will write `::_a` using infix notation. The following hh^2 definite clauses axiomatize the membership relation for type `a`.

```
memb X (X ::_a L).
memb X (Y ::_a L) :- memb X L.
```

With respect to the deterministic interpreter described above, these formulas implement the standard program testing list membership as often written in Prolog. In this case, the program is restricted to lists of type `a`. In the programs in this paper, we will make use of such a membership predicate at two primitive types. We will omit type subscripts from the constants `nil`, `::`, and `memb` since it will be clear from context which of the two is meant. In λProlog, these constants can be treated in a polymorphic fashion similar to their treatment in ML.

3 Specifying Sequent Calculi

Since we will be specifying first-order logic within the logic of hh^2, to avoid confusion we will refer to first-order logic as the *object-logic* to distinguish it from hh^2, the *meta-logic*. To represent first-order logic in hh^2, we introduce two primitive types: `form` for object-level formulas and `tm` for first-order terms. The new type `form` serves to distinguish formulas of the object-logic from formulas of the meta-logic (which have type `o`). Given these new primitive types, we introduce the following constants for the object-level connectives.

```
type    and     form -> form -> form.
type    or      form -> form -> form.
type    imp     form -> form -> form.
type    neg     form -> form.
type    forall  (tm -> form) -> form.
type    exists  (tm -> form) -> form.
type    false   form.
```

The constant `and`, for example, takes two formulas as arguments and constructs their conjunction. For readability, we write `and`, `or`, and `imp` using infix notation. By declaring `forall` and `exists` to take functional arguments, we have defined object-level binding of variables by quantifiers in terms of lambda abstraction, the meta-level binding operator, as is done in [1]. Thus, bound variables of the object-language are identified with bound variables of type `tm` at the meta-level. Meta-constants representing object-level constants, function symbols, propositions, and predicates can also be introduced and given appropriate types. For example, a constant `p` of type `tm -> form` represents an object-level unary predicate. Using these definitions, the first-order formula $\forall x \exists y (px \supset py)$, for instance, is represented by the λ-term below where `X` and `Y` are meta-variables of type `tm`.

```
(forall X\ (exists Y\ ((p X) imp (p Y))))
```

Figure 10 contains a complete set of inference rules for a sequent calculus for first-order intuitionistic logic. This inference system, which we call L_I, is a variant of the L system in [2]. In this system a sequent is written $\Gamma \longrightarrow A$ where Γ is a *set* of formulas, and A is a formula. Γ is called the *antecedent* of the sequent and A the *succedent*. Following convention, we write A, Γ to denote the set $\Gamma \cup \{A\}$. An *initial* sequent has the form $\Gamma \longrightarrow A$ where $A \in \Gamma$. The L_I inference system differs from the sequent system originally given in [6] in that we consider sets instead of sequences of formulas as antecedents and allow a more general form of initial sequent. As a result, in our formulation there is no need for explicit structural rules in the antecedent. The $\perp$-R rule is as in the specification of sequent systems in [15], and corresponds to the usual rule for thinning on the right.

To represent sets of formulas in antecedents, we will use lists of elements of type `form` where the order and number of copies of each element is not significant. Thus we introduce the primitive type `form_l`. In addition, we introduce the primitive type `seq` for sequents and add the signature item `-->` of type `form_l -> form -> seq`. We will write `-->` as an infix operator whose antecedent is a list of formulas and succedent is a single formula.

Finally, we introduce the primitive type `lprf` for the type of sequent proofs. The basic relation between a sequent and its proofs will be represented as a binary predicate at the meta-level by the infix constant `>-` of type `lprf -> seq -> o`. Each inference rule of the sequent calculus will be expressed as a simple declarative fact about this relation. Operationally, with respect to the interpreter described in the previous section, `>-` can be viewed as the theorem proving predicate.

For illustration purposes, we present the specification of a small subset of the inference rules of L_I. The others can be specified similarly. In specifying these rules, there are often many choices in choosing a representation of sequent proofs. In this presentation, we simply choose

$$\frac{\Gamma \longrightarrow A \qquad \Gamma \longrightarrow B}{\Gamma \longrightarrow A \wedge B}\wedge\text{-R} \qquad \frac{A, B, \Gamma \longrightarrow C}{A \wedge B, \Gamma \longrightarrow C}\wedge\text{-L}$$

$$\frac{\Gamma \longrightarrow A}{\Gamma \longrightarrow A \vee B}\vee\text{-R} \qquad \frac{\Gamma \longrightarrow B}{\Gamma \longrightarrow A \vee B}\vee\text{-R} \qquad \frac{A, \Gamma \longrightarrow C \qquad B, \Gamma \longrightarrow C}{A \vee B, \Gamma \longrightarrow C}\vee\text{-L}$$

$$\frac{A, \Gamma \longrightarrow B}{\Gamma \longrightarrow A \supset B}\supset\text{-R} \qquad \frac{\Gamma \longrightarrow A \qquad B, \Gamma \longrightarrow C}{A \supset B, \Gamma \longrightarrow C}\supset\text{-L}$$

$$\frac{A, \Gamma \longrightarrow \bot}{\Gamma \longrightarrow \neg A}\neg\text{-R} \qquad \frac{\Gamma \longrightarrow A}{\neg A, \Gamma \longrightarrow \bot}\neg\text{-L}$$

$$\frac{\Gamma \longrightarrow [y/x]A}{\Gamma \longrightarrow \forall x A}\forall\text{-R} \qquad \frac{[t/x]A, \Gamma \longrightarrow C}{\forall x A, \Gamma \longrightarrow C}\forall\text{-L}$$

$$\frac{\Gamma \longrightarrow [t/x]A}{\Gamma \longrightarrow \exists x A}\exists\text{-R} \qquad \frac{[y/x]A, \Gamma \longrightarrow C}{\exists x A, \Gamma \longrightarrow C}\exists\text{-L}$$

$$\frac{\Gamma \longrightarrow \bot}{\Gamma \longrightarrow A}\bot\text{-R} \qquad \frac{\Gamma \longrightarrow A \qquad A, \Gamma \longrightarrow C}{\Gamma \longrightarrow C}\text{cut}$$

The $\forall$-R and $\exists$-L rules have the proviso that the variable y cannot appear free in the lower sequent.

Figure 10: The L_I sequent calculus for first-order intuitionistic logic

one. For a more thorough discussion of the issues involved in specifying inference rules and proofs for this inference system and others in hh^2, see [5, 4].

First, consider the $\wedge$-R inference rule in Fig. 10 which introduces a conjunction on the right side of the sequent. This rule has a natural rendering as the following definite clause.

```
(and_r Q1 Q2) >- (Gamma --> (A and B)) :- Q1 >- (Gamma --> A),
                                          Q2 >- (Gamma --> B).
```

This formula may be read as: if `Q1` is a proof of `(Gamma --> A)` and `Q2` is a proof of `(Gamma --> B)`, then `(and_r Q1 Q2)` is a proof of `(Gamma --> (A and B))`. The constant `and_r` is a "proof constructor" which takes two proofs as arguments (the premises of the $\wedge$-R rule) and builds a new proof (its conclusion). Its type is `lprf -> lprf -> lprf`.

The following clause specifies the $\supset$-L rule and illustrates that introductions of logical constants into the antecedent of a sequent can be achieved similarly.

```
(imp_l (A imp B) Q1 Q2) >- (Gamma --> C) :- memb (A imp B) Gamma,
                                            Q1 >- (Gamma --> A),
                                            Q2 >- ((B::Gamma) --> C).
```

The main difference here is that the antecedent is a list instead of a single formula. We use the `memb` predicate (on type `form`) defined in Sect. 2 to choose an implication `(A imp B)` from the list `Gamma`. Note that the proof term in the head of the clause contains a record of the particular implication from `Gamma` to which the rule is applied.

We now consider the quantifier introduction rules. The declarative reading of the $\exists$-R inference rule is captured by the following definite clause.

```
(exists_r T Q) >- (Gamma --> (exists A)) :- Q >- (Gamma --> (A T)).
```

The existential formula of the conclusion of this rule is written `(exists A)` where the variable `A` has functional type `tm -> form`. Thus `A` is an abstraction over first-order terms and `(A T)` represents the formula that is obtained by substituting `T` for the bound variable in `A`. Declaratively, this clause reads: for first-order term `T`, if `Q` is a proof of `(Gamma --> (A T))`, then `(exists_r T Q)` is a proof of `(Gamma --> (exists A))`. Note that the proof term contains a record of the substitution term `T`. Operationally, the existential instance `(A T)` is obtained via the interpreter's operation of β-reduction. Thus β-conversion at the meta-level is used to specify substitution at the object-level.

Next, we consider the $\forall$-R rule, which has the additional proviso that y is not free in Γ or $\forall x A$. This proviso is handled by using a universal quantifier at the meta-level as in the following definite clause.

```
(forall_r Q) >- (Gamma --> (forall A)) :-
   pi Y\ ((Q Y) >- (Gamma --> (A Y))).
```

As in the previous clause, `A` has functional type. In this case, so does `Q`; it has type `tm -> lprf`, and thus the type of the constant `forall_r` is `(tm -> lprf) -> lprf`. Declaratively, this clause reads: if we have a function `Q` that maps arbitrary terms `Y` to proofs `(Q Y)` of the sequent `(Gamma --> (A Y))`, then `(forall_r Q)` is a proof of `(Gamma --> (forall A))`. Operationally, the GENERIC search operation is used to insert a new constant of type `tm` into the sequent. Since that constant will not be permitted to appear in `Gamma` or `A` the proviso will be satisfied.

As a final example, consider the cut rule which has the straightforward specification as the following formula.

```
(cut A Q1 Q2) >- (Gamma --> C) :- Q1 >- (Gamma --> A),
                                  Q2 >- ((A::Gamma) --> C).
```

A complete set of hh^2 formulas specifying the rules of L_I serves as a specification of a theorem prover for this inference system. To prove sequent `(Gamma --> A)`, we start with a goal of the form `(Q >- (Gamma --> A))` where `Q` is a logic variable to be filled in with a term representing a proof of `(Gamma --> A)`. Note, though, that there may be multiple definite clauses that can be used in backchaining to prove a particular sequent, or a single clause that can be used repeatedly. For example, if there is an implication in `Gamma`, it will always be possible to backchain on the clause specifying the $\supset$-L rule. Thus, this set of formulas cannot serve as an implementation with respect to a depth-first interpreter that backchains over definite clauses in a particular order. (See [5] for more on implementing theorem provers and controlling search in this setting.) On the other hand, when `Q` is a closed term in the original goal, this program behaves as a proof checker and is complete even with respect to the deterministic interpreter. The top-level constant of a proof term completely determines the unique definite clause to be used in backchaining at each step.

By Gentzen's cut-elimination result [6], L_I without the cut rule is a complete set of rules for first-order intuitionistic logic. Thus, by simply eliminating the formula for the cut rule, we obtain a set of hh^2 formulas which serves as a specification for both building and proof checking cut-free proofs.

$$\frac{A \qquad B}{A \wedge B}\ \wedge\text{-I} \qquad\qquad \frac{A \wedge B}{A}\ \wedge\text{-E} \qquad\qquad \frac{A \wedge B}{B}\ \wedge\text{-E}$$

$$\frac{A}{A \vee B}\ \vee\text{-I} \qquad\qquad \frac{B}{A \vee B}\ \vee\text{-I} \qquad\qquad \frac{A \vee B \qquad \begin{array}{c}(A)\\ C\end{array} \qquad \begin{array}{c}(B)\\ C\end{array}}{C}\ \vee\text{-E}$$

$$\frac{\begin{array}{c}(A)\\ B\end{array}}{A \supset B}\ \supset\text{-I} \qquad\qquad \frac{A \qquad A \supset B}{B}\ \supset\text{-E}$$

$$\frac{\begin{array}{c}(A)\\ \bot\end{array}}{\neg A}\ \neg\text{-I} \qquad\qquad \frac{A \qquad \neg A}{\bot}\ \neg\text{-E}$$

$$\frac{[y/x]A}{\forall x A}\ \forall\text{-I} \qquad\qquad \frac{\forall x A}{[t/x]A}\ \forall\text{-E}$$

$$\frac{[t/x]A}{\exists x A}\ \exists\text{-I} \qquad\qquad \frac{\exists x A \qquad \begin{array}{c}([y/x]A)\\ B\end{array}}{B}\ \exists\text{-E}$$

$$\frac{\bot}{A}\ \bot_I$$

The $\forall$-I rule has the proviso that the variable y cannot appear free in $\forall x A$, or in any assumption on which $[y/x]A$ depends.

The $\exists$-E rule has the proviso that the variable y cannot appear free in $\exists x A$, in B, or in any assumption on which the upper occurrence of B depends.

Figure 11: The N_I natural deduction inference system for first-order intuitionistic logic

4 Specifying Natural Deduction

Figure 11 presents the inference rules for natural deduction in intuitionistic logic as presented in [15], called N_I here. The premise of an elimination rule (E-rule) containing the connective for which the rule is named is called the *major premise*, and other premises are *minor premises*. The *discharge* of assumptions is indicated by parentheses. For example, in the $\supset$-I rule, (A) indicates that occurrences of A at the leaves are discharged by the application of this rule. A formula occurrence B in a tree is said to *depend* on an assumption A if A occurs as a leaf and is not discharged by a rule application above B. A tree with root B constructed using these inference rules is a *deduction* of B from the set of formulas Γ if all assumptions on which B depends occur in Γ. Such a tree is a *proof* of B if Γ is empty.

The inference rules of natural deduction can be specified directly in the same manner as those for the L_I sequent system, where the conclusion of the rule corresponds to the head of the clause and the premises to the subgoals. In the next subsection, we discuss this specification. The corresponding notion to cut-free proofs in L_I is *normal* proofs [15] in N_I. To specify N_I so that only normal proofs are built is more complicated than simply adding or removing clauses, which was all that was required to eliminate the use of cut in sequent proofs. Here, it is a matter of specifying the rules in a different manner. In subsection 4.2, we present such a specification. In subsection 4.3, we discuss alternative ways to specify some of the inference rules retaining the property that only normal proofs are built. The alternatives we present are in fact those needed to be able to merge the N_I specification with the cut-free L_I specification. Finally, in subsection 4.4, we present an alternative way of handling assumptions so that assumptions in N_I correspond to antecedents of sequents in L_I. The result is a specification that can be merged directly with the one for L_I discussed in the previous section.

4.1 A Direct Specification

To represent natural deduction proofs, we introduce the new primitive type `nprf`. Here, the basic proof relation is between proofs and formulas (instead of sequents). We introduce the infix predicate `#` of type `nprf -> form -> o` for this relation. In the previous section, we adopted the convention of using variable names `Q`, `Q1`, `Q2`, etc., to represent sequent proofs. We use `P`, `P1`, `P2`, etc., here for natural deduction proofs. Several of the introduction rules (I-rules) for this system resemble rules that apply to succedents in the sequential system just considered. The $\wedge$-I, $\exists$-I, and $\forall$-I rules correspond to examples given in the previous section and can be specified similarly as follows.

```
(and_i P1 P2) # (A and B) :- P1 # A, P2 # B.
(exists_i T P) # (exists A) :- P # (A T).
(forall_i P) # (forall A) :- pi Y\ ((P Y) # (A Y)).
```

Note that universal quantification is used to handle the proviso on the $\forall$-I rule in the same way that it is used for the $\forall$-R sequent rule.

In natural deduction, unlike sequential systems, we have the additional task of specifying the operation of discharging assumptions. Consider the implication introduction rule in Fig. 11. This rule can very naturally be specified as the definite clause below.

```
(imp_i P) # (A imp B) :- pi W\ ((W # A) => ((P W) # B)).
```

This clause represents the fact that if `P` is a "proof function" which maps an arbitrary proof `W` of formula `A`, to a proof `(P W)` of formula `B`, then `(imp_i P)` is a proof of `(A imp B)`. Here, the proof of an implication is represented by a function from proofs to proofs. The constant `imp_i` has the type `(nprf -> nprf) -> nprf`. Notice that while sequential proofs only contain abstractions of type `tm`, natural deduction proofs may contain abstractions of both types `tm` and `nprf`.

Operationally, when backchaining on this clause, the GENERIC operation is used to choose a new object, say `c`, to replace `W` and play the role of a proof of the formula `A`. The AUGMENT goal is then used to add the assumption `(c # A)` to the current set of program clauses. This clause is then available to use in the search for a proof of `B`, *i.e.*, in solving the goal `((P c) # B)`. The proof of `B` may contain instances of the proof of `A` (the constant `c`). The function `P` is the abstraction over this constant.

The elimination rules and $\perp$-I can be specified similarly to the introduction rules. For example the following formulas specify the $\wedge$-E and $\exists$-E rules.

```
(and_e1 B P) # A :- P # (A and B).
```

```
(and_e2 A P) # B :- P # (A and B).
(exists_e A P1 P2) # B :- P1 # (exists A),
   pi Y\ (pi W\ ((W # (A Y)) => ((P2 Y W) # B))).
```

The $\exists$-E rule contains both a proviso handled by a universal quantifier at the meta-level, and the discharge of an assumption, again handled by meta-level universal quantification and implication. Here, P2 is an abstraction over both the first-order term Y and the proof term W. Note that the proof terms in the heads of these clauses contain more than just the subproofs of the premises as arguments. For example, the missing conjunct in the conclusion of the $\wedge$-E rules is included in the proof terms for these rules.

As for L_I, the set of hh^2 formulas resulting from the direct specification of the rules of N_I serves both as a specification of a theorem prover and as an implementation of a proof checker with respect to the deterministic interpreter. As before, in proof checking, the top-level constant of a proof term completely determines the unique definite clause to be used in backchaining at each step.

4.2 A Specification of N_I That Constructs Normal Proofs

Before discussing specifications of N_I that construct normal proofs, several definitions are required. We begin by defining the notion of normal N_I proofs as in [15]. The main condition for an N_I deduction to be normal is that it must contain no *maximal formula*, that is, a formula that is the conclusion of an I-rule or $\perp_I$ and the major premise of an E-rule, since such applications are redundant (as in the example in Fig. 12 (a)). For the fragment of N_I without the $\vee$ and $\exists$ connectives, this condition is taken as the definition of normal. With these connectives in the logic the condition must be made slightly stronger, and requires some further definitions. A *segment* in a deduction is defined to be a sequence of formula occurrences $A_1, \ldots, A_n$ such that the following hold.

1. A_1 is not the conclusion of an application of $\vee$-E or $\exists$-E.

2. For $i = 1, \ldots, n-1$, A_i is a minor premise of an application of $\vee$-E or $\exists$-E.

3. A_n is not the minor premise of an application of $\vee$-E or $\exists$-E.

All occurrences in a segment are occurrences of the same formula. The deduction in Fig. 12 (b) contains a segment of length 3 of occurrences of $A \wedge B$. As in [15], we will say a segment is the premise of an application of a rule when its last formula is the premise of the rule. A *maximal segment* is a segment that begins with a conclusion of an application of an I-rule or $\perp_I$ and ends with a major premise of an E-rule. The segment of length 3 in Fig. 12 (b) is in fact maximal. Note that a maximal formula is a special case of a maximal segment. A *normal* deduction is then defined to be a deduction that contains no maximal segment. Figure 12 (c) contains a normal deduction of $r \supset s$ from $\{p \vee q, p \supset (r \supset s), q \supset (r \supset s)\}$.

Normal deductions can be characterized in terms of the form of certain sequences of formulas called paths. A *path* in a deduction is a sequence of occurrences $A_1, \ldots, A_n$ such that the following conditions hold.

1. A_1 is a leaf that is not discharged by an application of $\vee$-E or $\exists$-E.

2. For $i = 1, \ldots, n-1$, A_i is not the minor premise of an application of $\supset$-E or $\neg$-E. If A_i is the major premise of an application of $\vee$-E or $\exists$-E, then A_{i+1} is an assumption discharged by this application. Otherwise, A_{i+1} is the conclusion of the rule for which A_i is a premise.

$$\dfrac{\dfrac{A \qquad B}{A \wedge B}\,\wedge\text{-I}}{A}\,\wedge\text{-E}$$

(a)

$$\dfrac{\dfrac{\exists y D \qquad \dfrac{\exists x C \qquad \dfrac{A \qquad B}{A \wedge B}\,\wedge\text{-I}}{A \wedge B}\,\exists\text{-E}}{A \wedge B}\,\exists\text{-E}}{A}\,\wedge\text{-E}$$

(b)

$$\dfrac{\dfrac{r \qquad \dfrac{p \vee q \qquad \dfrac{p \quad p \supset (r \supset s)\ {\scriptstyle (1)}}{r \supset s\ {\scriptstyle (2)}}\,\supset\text{-E} \qquad \dfrac{q \quad q \supset (r \supset s)}{r \supset s}\,\supset\text{-E}}{r \supset s\ {\scriptstyle (2)}}\,\vee\text{-E}}{s\ {\scriptstyle (3)}}\,\supset\text{-E}}{r \supset s\ {\scriptstyle (4)}}\,\supset\text{-I}$$

(c)

Figure 12: Maximal formulas, maximal segments, and paths in N_I deductions

3. A_n is either the minor premise of $\supset$-E or $\neg$-E or the root of the deduction.

A path of length 5 (containing 4 segments) in a normal deduction is indicated in Fig. 12 (c). Note that the second segment, indicated by the number (2), has length 2. The other paths in this deduction are:

1. a similar path starting with $q \supset (r \supset s)$
2. $p \vee q, p$
3. $p \vee q, q$
4. r.

We define an *E-path* to be a subsequence of a path such that each segment except the last is a major premise of an E-rule, and an *I-path* to be a subsequence of a path such that each segment except the first is a conclusion of an I-rule or $\perp_I$. We say that a deduction is an *E-deduction* if all paths containing the root are E-paths. Note that the subtree rooted at s in Fig. 12 (c) is an E-deduction.

In a normal deduction, each path contains a series of segments divided into two parts: an E-path followed by an I-path such that the last segment in the E-path called the *minimum segment* is also the first segment in the I-path. In Fig. 12 (c), the formula occurrence s is the minimum segment separating the E-paths and the I-paths of the two paths beginning with $p \supset (r \supset s)$ and $q \supset (r \supset s)$. The minor premises and conclusion of applications of $\vee$-E and $\exists$-E may appear in the E-path or I-path (or both) of all paths through them. In Fig. 12 (c), the middle premise and conclusion of $\vee$-E, for example, occur in the E-path of the indicated path. Note that an E-deduction is normal if and only if all deductions rooted at a minor premise of an application of $\supset$-E or $\neg$-E are normal.

The division of paths in normal deductions into E-paths and I-paths will be reflected in our specification. We will use two relations in specifying the definite clauses for the inference rules. The first one is the `#e` predicate used to relate a formula and a normal E-deduction. For the second, we continue to use #, but this time to relate a formula with a normal deduction. Both predicates have type `nprf -> form -> o`. Thus, in a provable formula of the form `(P #e A)`, `P` represents a normal E-deduction of `A`, while in a provable formula of the form `(P # A)`, `P` is a normal deduction of `A`. Operationally, the clauses for the `#e` relation will apply E-rules, and the clauses for the `#` relation will apply I-rules and join I-paths and E-paths at the minimum segment.

Using these two predicates, the introduction rules and $\perp_I$ can be specified exactly as in the direct specification, except that discharged assumptions are added as facts about the `#e` relation since they are one-node E-deductions. They may occur at the leaves in larger deductions and will always occur in the E-paths that pass through them. Thus, the $\supset$-I rule, for instance, is now specified as follows.

```
(imp_i P) # (A imp B) :- pi W\ ((W #e A) => ((P W) # B)).
```

The elimination rules $\wedge$-E, $\supset$-E, $\neg$-E, and $\forall$-E are specified using the `#e` predicate since, in a normal deduction, both the major premise and conclusion of all applications of these rules are roots of normal E-deductions. For example, the clauses for the $\wedge$-E rules now have the following form.

```
(and_e1 B P) #e A :- P #e (A and B).
(and_e2 A P) #e B :- P #e (A and B).
```

The $\supset$-E rule, on the other hand, is represented by the following clause where the subgoal for the minor premise uses the `#` predicate.

```
(imp_e A P1 P2) #e B :- P1 # A, P2 #e (A imp B).
```

This reflects the fact that the subproof at the minor premise can be an arbitrary normal deduction. By the definition of path, the root of this subproof will always be the last formula occurrence in the paths through it. The $\neg$-E rule is similar.

$\vee$-E and $\exists$-E are each specified by two definite clauses, the first corresponding to when the minor premises and conclusion occur in I-paths of paths through them, and the second when they occur in E-paths. The following two clauses are those for the $\vee$-E rule.

```
(or_e A B P P1 P2) # C :- P #e (A or B),
                          pi W\ ((W #e A) => ((P1 W) # C)),
                          pi W\ ((W #e B) => ((P2 W) # C)).

(or_e A B P P1 P2) #e C :- P #e (A or B),
                           pi W\ ((W #e A) => ((P1 W) #e C)),
                           pi W\ ((W #e B) => ((P2 W) #e C)).
```

Note that when the minor premises of this rule are the roots of arbitrary normal deductions, the conclusion is also at the root of an arbitrary normal deduction, as indicated by the use of the `#` relation to relate formula `C` to its various proofs in the head and body of the first of the two clauses above. On the other hand, when the minor premises of this rule are at the roots of normal E-deductions, as in the latter clause above, the conclusion is also the root of a normal E-deduction.

Finally, we must also add the clause `P # A :- P #e A`. This clause serves to join I-paths and E-paths at the minimum segment. Its declarative reading is that any normal E-deduction is a normal deduction.

4.3 Alternative Specifications for Constructing Normal Proofs

We now discuss several modifications that can be made to this specification. This presentation both illustrates alternative ways of specifying the rules of N_I so that only normal proofs get constructed and also brings us closer to a specification that corresponds to cut-free L_I.

In the specification discussed in the previous subsection, the $\neg$-E rule would have the following formulation, since it is similar to $\supset$-E.

```
(neg_e A P1 P2) #e false :- P1 # A, P2 #e (neg A).
```

In a normal N_I deduction, any occurrence of $\perp$ is always in the minimum segment of paths through it, since it cannot be the major premise of an E-rule or the conclusion of an I-rule. As a result, we have the option of modifying the above formula so that `#` instead of `#e` appears in the head of the clause. With this modification, it is still the case that `P` in a provable formula of the form `(P #e A)` represents an E-deduction of `A`, but now it is one that does not end in $\neg$-E. Operationally, the clauses for the `#` relation now build I-paths in a goal directed fashion, and possibly also add the last segment in the E-paths of paths through `false`. Clauses for `#e` will then add the remaining E-paths.

We can simplify the specification of the previous subsection if we consider the following refinement of normal deductions. We define an *E-segment* in a normal deduction to be a segment whose last occurrence is the conclusion of an application of $\vee$-E or $\exists$-E and the major premise of an E-rule. Note that maximal segments are a special case of E-segments. As pointed out to Prawitz by Martin-Löf [14], the definition of normal can be sharpened to require that normal deductions contain no E-segments or maximal formulas. We call such deductions *S-normal* deductions. In an S-normal deduction, every minor premise or conclusion of an application of $\vee$-E and $\exists$-E will appear either in the minimum segment (in which case it is both in the I-path and E-path) or only in the I-path of paths through it. The deduction below is a modification of the deduction in Fig. 12 (c) that meets the extra restriction on segments to make it S-normal.

$$\cfrac{p \vee q \qquad \cfrac{\cfrac{r \qquad \cfrac{p \qquad p \supset (r \supset s)}{r \supset s}\,\supset\text{-E}}{s}\,\supset\text{-E}}{r \supset s}\,\supset\text{-I} \qquad \cfrac{\cfrac{r \qquad \cfrac{q \qquad q \supset (r \supset s)}{r \supset s}\,\supset\text{-E}}{s}\,\supset\text{-E}}{r \supset s}\,\supset\text{-I}}{r \supset s}\,\vee\text{-E}$$

The specification for N_I that we will eventually merge with the specification for L_I is one that builds deductions in this sharpened normal form. In order to discuss this specification we refine some definitions given earlier. By an *E-path*, we will still mean a subsequence of a path such that each segment except the last is a major premise of an E-rule, but now we add the additional restriction that each segment must be of length 1. The definition of I-path remains the same. In an E-deduction, all paths containing the root must now meet the additional restriction on E-paths. With respect to this modified definition, each path in an S-normal deduction will still contain an E-path followed by an I-path, but now there may not be any overlap in these two parts. The minimum segment will always be in the I-path, but will not be in the E-path if it has length greater than 1.

Note that under these new definitions, the latter of the two clauses given for $\vee$-E in the previous subsection is no longer correct. Even when all three premises are E-deductions, the conclusion will not be, since the root will be in a segment of length greater than 1, and thus paths through it will not be E-paths. Hence, we must eliminate this clause and the similar one for $\exists$-E. The simple elimination of these two clauses is all that is required to obtain a specification that constructs only S-normal deductions. In the remainder of this paper, we use the term "normal" to mean "S-normal" since we now only consider deductions in S-normal form.

With the above modifications, the remaining clauses with `#e` in the head are those for $\wedge$-E, $\supset$-E, and $\forall$-E. These rules can also be modified. We illustrate using the $\wedge$-E rules. The clauses in subsection 4.2 specifying the $\wedge$-E rules can be replaced by the following clause with `#` in the head.

```
PC # C :- P #e (A and B),
          (((and_e1 B P) #e A) => (((and_e2 A P) #e B) => (PC # C))).
```

The declarative reading of this clause is as follows: `PC` is a proof of formula `C` if there is a normal E-deduction of conjunct `(A and B)` (whose proof is `P`), and from the assumptions that `A` and `B` are provable separately (with proofs `(and_e1 B P)` and `(and_e2 A P)` respectively) it can be shown that `PC` is a proof of `C`. Operationally, in attempting to find a normal deduction for any formula `C`, this clause will look for an E-deduction of a conjunction `(A and B)`, apply both versions of the $\wedge$-E rule to it to obtain two new E-deductions, add the new subproofs as atomic program clauses, and attempt to find a normal deduction for `C` in the environment extended with these new assumptions.

As stated and proved in [15], normal deductions have the *subformula property*, that is, every formula occurring in a normal deduction of A from Γ is a subformula of A or of some formula in Γ. In fact, every formula occurring in an E-path is a subformula of a formula in Γ, every formula occurring in an I-path is a subformula of A, and every formula occurring in a minimum segment is a subformula of both A and some formula in Γ. This property is reflected in the following operational description of our new specification. The clauses for the I-rules apply the rules in a backward direction so that the formulas in the subgoals (which correspond to the premises) are always subformulas of the formula in the head of the clause (the conclusion). In contrast, the clauses for $\wedge$-E, $\supset$-E, and $\forall$-E apply the rules in a forward direction from the assumptions so that the conclusion is always a subformula of the major premise. In applying E-rules, new E-deductions get built from existing E-deductions and are then added to the program as new facts about the `#e` relation.

Note that this program can no longer serve as a proof checker with respect to the deterministic interpreter. It is no longer the case that the top-level constant of a proof term completely determines the definite clause to be used in backchaining. In fact, clauses like the specification of $\wedge$-E above can always be used in attempting to prove an atomic goal for the `#` predicate since any formula and proof term will be instances of the formula and proof in the head of the clause.

4.4 Explicit vs. Implicit Representation of Assumptions in Natural Deduction

In specifying N_I we showed that it was quite natural to specify the discharge of assumptions "implicitly" using universal quantification and implication at the meta-level. It is also possible to explicitly keep track of assumptions by storing them in a list, and make use of a membership predicate to extract individual elements in much the same way that antecedents were handled in the specification of L_I. For N_I, such assumption lists will contain pairs of formulas associated with their E-deductions. We will call such lists of pairs *contexts*. We can obtain an "explicit context" specification of N_I via a systematic modification of the definite clauses of any of the "implicit context" specifications discussed so far. We illustrate using the formulation in the previous subsection.

We continue to use `#e` and `#` for the basic relations between a formula and its deductions, but no longer as predicates; they will now have the type `nprf -> form -> judg` where `judg` is a new primitive type introduced to represent these basic judgments. Contexts will be lists of elements of type `judg_l` where `judg_l` is, of course, the primitive type introduced for lists of elements of type `judg`. We must modify each definite clause so that it has an extra argument for a context, which corresponds to the set of assumptions that exist at the time the rule is applied. We use the sequent arrow `-->` as our predicate, now used to form "judgment sequents." This predicate has type `judg_l -> judg -> o` and expresses the relation between a context and a single formula paired with a normal deduction or normal E-deduction.

For those clauses that do not involve the use of implication to add assumptions to the program, the modification simply requires adding a list and sequent arrow to form a judgment sequent in the head and subgoals of each clause. For example, the clause for $\wedge$-I is rewritten as follows.

```
Gamma --> ((and_i P1 P2) # (A and B)) :- Gamma --> (P1 # A),
                                         Gamma --> (P2 # B).
```

The discharge of assumptions as in the $\supset$-I rule is specified as below where the new assumption gets added to the context rather than the program.

```
Gamma --> ((imp_i P) # (A imp B)) :-
   pi W\ (((W #e A)::Gamma) --> ((P W) # B)).
```

All other formulas can be modified similarly. For example, the clause for $\wedge$-E adds two assumptions to the context.

```
Gamma --> (PC # C) :- Gamma --> (P #e (A and B)),
   (((and_e1 B P) #e A)::((and_e2 B P) #e B)::Gamma) --> (PC # C).
```

The formula expressing the fact that a normal E-deduction is a normal deduction is replaced by the following clause.

```
Gamma --> (P # A) :- Gamma --> (P #e A).
```

As before, this clause operationally joins E-paths and I-paths at the minimum segment. Finally, we need the following clause for completing E-deductions.

```
Gamma --> (P #e A) :- memb (P #e A) Gamma.
```

Of course, the `memb` predicate here is at type `judg`. In previous implicit context specifications, we did not need an explicit clause for closing E-deductions. There, an E-deduction was closed by unifying an atomic goal formula of the form `(P #e A)` with an atomic clause of the program.

This completes the description of the systematic modification from implicit to explicit context specifications. We now discuss two final modifications that can be made to this particular specification of N_I.

Recall that in the specification in the previous subsection, all clauses have `#` as the predicate occurring in the head. Thus in the modified specification here, all clauses except for the one immediately above have `#` as the relation on the right of the sequent arrow in the head of the clause. Hence, whenever an atomic formula of the form `(Gamma --> (P #e A))` succeeds, it must be the case that (P #e A) is in the context Gamma. We could in fact remove the clause above for closing E-deductions, and replace each subformula of the form (Gamma --> (P #e A)) with (memb (P #e A) Gamma) directly in the formulas specifying the inference rules. The formula for $\wedge$-E would then be written as follows.

```
Gamma --> (PC # C) :- memb (P #e (A and B)) Gamma,
   (((and_e1 B P) #e A)::((and_e2 A P) #e B)::Gamma) --> (PC # C).
```

After making such a modification, the resulting set of formulas is such that only the # relation appears on the right, while only the #e relation appears on the left of the sequent arrow. Thus, we can distinguish normal E-deductions from arbitrary normal deductions simply by where they occur in judgment sequents. As a result, we no longer need two distinct relations. They can be merged into one. We do so here by simply eliminating the #e relation and adopting # as the single relation between a formula and a deduction. Using this formulation, the above clause for $\wedge$-E becomes the following formula.

```
Gamma --> (PC # C) :- memb (P # (A and B)) Gamma,
   (((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> (PC # C).
```

The specification obtained by making similar modifications to all of the clauses is the one we will now directly merge with the L_I specification.

5 Transforming L_I Proofs to N_I Proofs

The first step in merging the specifications of L_I and N_I is to combine the data structures for sequents and judgments. We do so using the same constants as before, but now give them the following types.

```
type    #       nprf -> form -> judg.
type    -->     judg_l -> judg -> seq.
type    >-      lprf -> seq -> o.
```

As before # is used for the relation between a formula and N_I deduction. We also again use the sequent arrow for judgment sequents. Here the type seq replaces o since it will no longer serve as the top-level predicate. As in the specification of L_I, >- is the top-level predicate with the same type as before. Note, though, that the second argument is a judgment sequent in this case. An atomic formula now has the form (Q >- (Gamma --> (P # A))) where Gamma is a list of judgment pairs. If such a formula is provable, it will be the case that Q represents a cut-free L_I proof of the sequent whose antecedent contains the formulas in Gamma and whose succedent is A, and P represents a normal N_I deduction of A from the formulas in Gamma.

In the clauses in this specification, the L_I rules for introducing a connective on the right of the sequent will be simultaneously specified with the corresponding N_I introduction rule. Similarly, the rules for introducing a connective on the left in L_I will be specified with the corresponding elimination rule. Finally, the $\perp$-R rule in L_I and the $\perp_I$ rule in N_I will be specified together. For example, the first formula below simultaneously specifies the $\wedge$-R and $\wedge$-I rules from L_I and N_I, respectively, while the second formula specifies both $\wedge$-L and $\wedge$-E.

```
(and_r Q1 Q2) >- (Gamma --> ((and_i P1 P2) # (A and B))) :-
   Q1 >- (Gamma --> (P1 # A)), Q2 >- (Gamma --> (P2 # B)).
```

```
(and_l (A and B) Q) >- (Gamma --> (PC # C)) :-
   memb (P # (A and B)) Gamma,
   Q >- ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> (PC # C)).
```

They are obtained by a straightforward merging of the corresponding formulas of the separate specifications. They illustrate how N_I proof terms are associated with formulas within a sequent on the right and left, respectively, while the L_I proof terms are associated with the entire sequent using the top-level relation.

The following clause simultaneously specifies ∀-R and ∀-I.

```
(forall_r Q) >- (Gamma --> ((forall_i P) # (forall A))) :-
   pi Y\ ((Q Y) >- (Gamma --> ((P Y) # (A Y)))).
```

This clause illustrates how universal quantification at the meta-level is used to handle simultaneously the provisos on both ∀-R and ∀-I. Note that both `Q` and `P` are abstractions over the variable `Y`.

As a final example, consider the formula below for ⊃-R and ⊃-I.

```
(imp_r Q) >- (Gamma --> ((imp_i P) # (A imp B))) :-
   pi W\ (Q >- (((W # A)::Gamma) --> ((P W) # B))).
```

Universal quantification is used to introduce a constant to replace the variable `W` and serve as a proof for hypothesis `A`. As before, the term `P` is an abstraction over this constant. It represents a function from proofs of `A` to proofs of `B`. Note, on the other hand that, as in the L_I specification, `Q` is not an abstraction.

The complete specification for L_I and N_I is given in Fig. 13. Operationally, this program can take on several roles. In a query of the form:

```
(Q >- (nil --> (P # A)))
```

where both `Q` and `P` are logic variables, the program behaves as a theorem prover, and simultaneously constructs proofs in both inference systems. If `P` is also specified, then the program acts as a proof transformer, transforming an N_I proof `P` to an L_I proof `Q`. Conversely, an L_I proof `Q` can be used to guide the construction of an N_I proof `P`. The program is complete with respect to the deterministic depth-first interpreter for this latter transformation, for the same reason that the L_I specification is complete as a proof checker. The constant at the head of the proof term `Q` uniquely determines which definite clause must be used at each step. The converse, however, is not true since as discussed in subsection 4.3, this formulation of the N_I rules cannot serve as a proof checker with respect to the deterministic interpreter because of the clauses for the ∧-E, ⊃-E, and ∀-E rules. If however, the depth-first search strategy of the interpreter were replaced by breadth-first search, the transformation in this direction would be achieved by this program.

Alternatively, we could modify the specification in Fig. 13 so that the transformation from N_I to L_I proofs works with respect to the depth-first interpreter. One simple approach involves taking into account the form of E-deductions in S-normal proofs. In such deductions, there is one path to the root that begins at an assumption and contains a sequence of applications of ∧-E, ⊃-E, ∀-E, possibly ending in an application of ¬-E. One or more of the sequent proofs to which such an E-deduction is related will contain a sequence of ∧-L, ⊃-L, and ∀-L rules occurring in reverse order to the corresponding I-rules in the natural deduction proof. It is straightforward to

```
(initial A) >- (Gamma --> (P # A)) :- memb (P # A) Gamma.

(and_r Q1 Q2) >- (Gamma --> ((and_i P1 P2) # (A and B))) :-
   Q1 >- (Gamma --> (P1 # A)), Q2 >- (Gamma --> (P2 # B)).

(or_r1 Q) >- (Gamma --> ((or_i1 P) # (A or B))) :-
   Q >- (Gamma --> (P # A)).

(or_r2 Q) >- (Gamma --> ((or_i2 P) # (A or B))) :-
   Q >- (Gamma --> (P # B)).

(imp_r Q) >- (Gamma --> ((imp_i P) # (A imp B))) :-
   pi W\ (Q >- (((W # A)::Gamma) --> ((P W) # B))).

(neg_r Q) >- (Gamma --> ((neg_i P) # (neg A))) :-
   pi W\ (Q >- (((W # A)::Gamma) --> ((P W) # false))).

(forall_r Q) >- (Gamma --> ((forall_i P) # (forall A))) :-
   pi Y\ ((Q Y) >- (Gamma --> ((P Y) # (A Y)))).

(exists_r T Q) >- (Gamma --> ((exists_i T P) # (exists A))) :-
   Q >- (Gamma --> (P # (A T))).

(false_r Q) >- (Gamma --> ((false_i P) # A)) :-
   Q >- (Gamma --> (P # false)).

(and_l (A and B) Q) >- (Gamma --> (PC # C)) :-
   memb (P # (A and B)) Gamma,
   Q >- ((((and_e1 B P) # A)::((and_e2 A P) # B)::Gamma) --> (PC # C)).

(imp_l (A imp B) Q1 Q2) >- (Gamma --> (PC # C)) :-
   memb (P2 # (A imp B)) Gamma, Q1 >- (Gamma --> (P1 # A)),
   Q2 >- ((((imp_e A P1 P2) # B)::Gamma) --> (PC # C)).

(forall_l (forall A) T Q) >- (Gamma --> (PC # C)) :-
   memb (P # (forall A)) Gamma,
   Q >- ((((forall_e A T P) # (A T))::Gamma) --> (PC # C)).

(neg_l (neg A) Q) >- (Gamma --> ((neg_e A P1 P2) # false)) :-
   memb (P2 # (neg A)) Gamma, Q >- (Gamma --> (P1 # A)).

(or_l (A or B) Q1 Q2) >- (Gamma --> ((or_e A B P P1 P2) # C)) :-
   memb (P # (A or B)) Gamma,
   pi W\ (Q1 >- (((W # A)::Gamma) --> ((P1 W) # C))),
   pi W\ (Q2 >- (((W # B)::Gamma) --> ((P2 W) # C))).

(exists_l (exists A) Q) >- (Gamma --> ((exists_e A P1 P2) # B)) :-
   memb (P1 # (exists A)) Gamma,
   pi Y\ (pi W\ ((Q Y) >- (((W # (A Y))::Gamma) --> ((P2 Y W) # B)))).
```

Figure 13: Definite clauses for simultaneous specification of L_I and N_I

write a program to convert an E-deduction to such a corresponding "inverted" sequent proof. Then, a program to convert general N_I normal deductions to L_I proofs can be obtained by removing the clauses for the $\wedge$-E, $\supset$-E, and $\forall$-E rules from the clauses in Fig. 13 and modifying the remaining program to make use of such an inversion procedure for E-deductions.

6 Conclusion

The program we have presented provides both a declarative and an operational description of the correspondence between the sequent calculus and natural deduction for intuitionistic logic. An interesting next step would be to extend this approach to provide illustrations of other well-known proof-theoretical results. One obvious candidate is the correspondence between cut-elimination in sequent systems and proof normalization in natural deduction. Such a specification may provide a program that simultaneously performs both operations.

In Sect. 4.4, we illustrated how to convert the specification of Sect. 4.3, which uses meta-level implication for the discharge of assumptions, into a specification using explicit assumption lists. It is easy to see that this operation can be performed on any of the "implicit context" specifications of natural deduction presented here. In fact, this operation is not limited to an intuitionistic object-logic. Specifications of natural deduction for classical logic or higher-order logic, for example, could be similarly transformed. Any such explicit context specification can be viewed as a specification of a sequent style inference system for the same logic. In the case of N_I, the fact that the inference rules could be specified in such a way that the corresponding sequent system was exactly L_I without the cut rule is a result of the close correspondence between the two. The well-known sequent and natural deduction inference systems for classical logic, for example, cannot be so easily related in this way.

Acknowledgements

I am grateful to Dale Miller and Peter Schroeder-Heister for valuable discussions and comments related to this paper. This research was supported in part by grants ARO-DAA29-84-9-0027, ONR N00014-88-K-0633, NSF CCR-87-05596, DARPA N00014-85-K-0018, and ESPRIT Basic Research Action 3245.

References

[1] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[2] Michael Dummett. *Elements of Intuitionism.* Clarendon Press, Oxford, 1977.

[3] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. May 1989.

[4] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, University of Pennsylvania, August 1989.

[5] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, Argonne Ill., May 1988.

[6] Gerhard Gentzen. Investigations into logical deductions, 1935. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., Amsterdam, 1969.

[7] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus.* Cambridge University Press, 1986.

[8] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[9] 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.*

[10] Gopalan Nadathur and Dale Miller. Higher-order horn clauses. April 1988. To appear in the *Journal of the ACM.*

[11] 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.

[12] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

[13] Garrel Pottinger. Normalization as a homomorphic image of cut-elimination. *Annals of Mathematical Logic*, 12(3):223–357, 1977.

[14] 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.

[15] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.

[16] J. I. Zucker. Cut-elimination and normalization. *Annals of Mathematical Logic*, 1(1):1–112, 1974.

The Gilbreath Trick:
A case study in Axiomatisation and Proof Development in the Coq Proof Assistant

Gérard Huet
INRIA*

June 1991

Introduction

This note presents a complete formal proof, in version 5.6 of the Proof Assistant Coq, of a card trick due to Norman Gilbreath[15, 13, 14], and shown to the author by N. G. de Bruijn[5]. The trick is based on a mathematical theorem concerning the shuffles of alternated binary words. A complete formalisation of the problem is given below. Coq is an implementation of the Calculus of Inductive Constructions[11] under development at INRIA in Rocquencourt and at ENS in Lyon[12].

This note is basically a transcript of what the user should type in order to mechanically verify the proof. Coq uses a top-down proof engine which solves successive subgoals with tactics in the style of LCF[16]. In the transcript below we do not systematically show the output of the system, which lists at every step the remaining subgoals. The proof is thus fully understandable only if one runs this proof interactively with our system. Furthermore, the transcript does not faithfully represent the proof-finding activity, with its numerous erroneous attempts, and the top-down development leaving to future proofs mathematical facts which are initially assumed as axioms. Finally, this transcript does not give a feeling for the actual interactive use of our system, which may be optionally driven from a sophisticated window interface. However, we hope that this commented example will be of use to others, both in order to assess the present capabilities of our system, and in order to get some familiarity with our axiomatisation style and proof methodology.

1 Axiomatising mathematics in Coq

Every mathematical object defined or axiomatised in the system is equipped with its *type*. We also say that the object *inhabits* its type. There are two *sorts* of types: propositions and sets. Propositions are of sort *Prop*, sets are of sort *Set*. We write logical propositions for the statements of axioms and theorems. We write set specifications for the definition of mathematical notions. Sets are inhabited by their elements. Provable propositions are inhabited by proof objects, which are of no concern to the naive user.

*This research was partially supported by ESPRIT Basic Research Action "Logical Frameworks."

Objects and types may be bound to a name through a definition. Names may also be declared with a type declaration, either with a global parameter declaration (for instance, axioms are names declared with a proposition as type), or with a local binding operator.

The main constructions may be briefly described as follows:

- names refer to declared variables or defined constants
- `(M N)` denotes the application of functional object M to object N.
- `[x:T]M` abstracts variable x of type T in construction M in order to construct a functional object $\lambda x \in T \cdot M$.
- `(x:T)P` as set specification, corresponds to a product type $\Pi x \in T \cdot P$, abbreviated as an arrow type $T \rightarrow P$ when x does not occur in P. As a proposition, it corresponds to universal quantification, i.e. $\forall x \in T \cdot P$. The arrow abbreviation in this case corresponds to logical implication.
- `Ind{X:S|C1; ... Cn}` defines inductively a set or a proposition, according to sort S, with n *constructors* of types C_1, ..., C_n.

Further cases refer to the constructors of an inductive type, and its elimination principles, corresponding to an induction principle for propositions, and a recursion principle for sets. These constructions are rarely used directly by the user. A higher level language, called the *mathematical vernacular*, permits to name these notions conveniently. For instance, the inductive set of natural numbers is described as:

```
Inductive Set nat = O : nat | S : nat -> nat.
```

This defines together the inductive set, bound to name `nat`, its two constructors, bound to names `O` and `S`, an induction principle of type the proposition:

```
(P:nat->Prop)(P O)->((x:nat)(P x)->(P (S x)))->(n:nat)(P n)
```

and a recursion principle, of type the specification:

```
(P:nat->Set)(P O)->((x:nat)(P x)->(P (S x)))->(n:nat)(P n)
```

A typical inductive predicate definition (here $\leq$) will look like:

```
Inductive Definition le [n:nat] : nat -> Prop =
      le_n : (le n n)
    | le_S : (m:nat)(le n m)->(le n (S m)).
```

Such a definition may be thought of as a (typed) Prolog specification. We may also write recursive definitions in a functional style:

```
Definition plus = [n,m:nat](<nat>Match n with
                                    (* O *)     m
                                    (* (S p) *) [p:nat]S).
```

which corresponds to the usual recursion equations for addition (the strings `(* ... *)` are just comments in this syntax).

The standard logical connectives, equality, and existential quantification, are definable as inductive notions. Indeed, the standard prelude of the system declares these basic notions, with their usual syntax. For instance, `P /\ Q` (resp. `P \/ Q`) expresses the conjunction (resp. disjunction) of propositions P and Q. The term `<nat>n=m` expresses equality of naturals n and m. It is actually shorthand for `(eq nat n m)`, where *eq* is the inductively-defined equality relation. In this case the unique constructor is reflexivity of equality, and the induction principle corresponds to the replacement of equals by equals. Such internal codings may be ignored in first approximation.

Thus, a typical mathematical axiomatisation in Coq will consist of a mixture of (higher-order) predicate calculus, non-deterministic specifications in the style of logic-programming, and recursive definitions in the style of functional-programming. Type-checking is done modulo definitional equality, which corresponds to replacing a defined constant by its definition, parameter substitution in functional applications (i.e. β-reduction in the underlying λ-calculus), and recursion unfolding (i.e. applying simplifications such as $(plus\ O\ m) \Rightarrow m$).

This blend of high-level notations leads to a smooth style of axiomatisation, relatively easy to master by anyone with a minimal training in formal methods, without requiring a complete understanding of the logical framework of the Calculus of Inductive Constructions.

2 A brief description of the theorem prover

Coq uses a top-down theorem prover which solves successive subgoals with tactics in the style of LCF[16]. The user formulates a conjecture with the command `Goal P`, where P is a well-formed proposition in the current context. The system then enters a proof-development mode, with a local top-level loop which at every step prints the remaining subgoals, and waits for a prover command in a syntax of tactics invocation. We shall briefly describe in this section the five families of tactics which are used in the proof below. It is also possible to interact with the prover through an interactive interface, displaying subgoals in windows on the screen of the user's workstation, and driven by user clicks on mouse-sensitive buttons. A full manual of the system is available in the standard distribution[12].

The first class of tactics concerns parameter introductions. The tactic `Intro H`, applied to a product goal of the form $A \rightarrow B$, enters $H : A$ in the current signature, or local context, and returns B as a subgoal. If the product is dependent, like $(x : A)(P\ x)$ (read $\forall x \in A \cdot P(x)$), we subgoal into $(P\ H)$. If the argument H is missing, the system will generate an appropriate name. If H exists already as a local hypothesis, the system complains. The tactic `Intros` iterates `Intro`, either with explicitly given names separated with blanks, or else as long as the current goal is a product.

The second class is that of resolution tactics. The tactic `Apply` tries to resolve the current goal with its argument, applied to terms, whose types become new goals to be solved. This corresponds to the refinement tactic of Nuprl[6]. This tactic may be helped by giving explicit instanciations, like in `Apply Lemma1 with (f x)`. Special cases of `Apply` are called `Left` (resp. `Right`) for the application of propositional *or* introduction to the left (resp. right). Also `Split` corresponds to *and* introduction, by splitting a goal of the form $A \wedge B$ into subgoals A and B.

The third class is the elimination tactics. They are the work horse of induction proofs. `Elim T`, when the term `T` has inductive type, does case analysis on the different constructors of this type. For instance, `Elim x`, when we have an hypothesis $x : A \wedge B$, will transform the current goal G into $A \rightarrow B \rightarrow G$. The `Induction` tactic combines introduction and elimination. Thus `Induction y` on a goal $(x : A)(y : B)(R\ x\ y)$ is equivalent to `Intros x y` followed by `Elim y`. The notation `Induction 2`, for instance, means do induction on the second un-named current hypothesis. A special tactic permits to do proofs by contradiction as follows. `Absurd P` replaces the current goal by the two goals P and $\sim P$. It is a special case of elimination of the absurd proposition, noted `{}`, combined with application of negation.

The fourth class is that of equality tactics. The `Simpl` tactics simplifies the current goal by putting it in normal form. This combines β-reduction of λ-calculus redexes, as well as primitive recursion on inductive types. The tactic `Unfold` unfolds the definition of the constant given as its argument. It applies to the first occurrence of the given constant in the current goal. If one wants to unfold the (say) third occurrence of constant C, it is possible to do it with `Unfold 3 C`. `Unfold * C` unfolds all the occurrences of constant C. Finally, `Change T` changes the current goal G by term T, after checking that G and T are interconvertible terms.

The fifth class comprises tactics which do some limited automatic search: The tactic `Trivial` tries to find the current goal either in the local hypotheses, or in a list of simplification axioms and lemmas. One enters a new such simplification with the tactic `Hint`. For instance, applications of reflexivity of equality are handled by `Trivial`. A more sophisticated theorem-proving tactic is the tactic `Auto`, which iterates such simplification to a nesting depth of 5; it implements a limited form of automated deduction.

Tactics may be composed with the THEN tactical, concretely written as a semi-colon. Thus a command line is usually of the form: `Tactic 1; Tactic 2; ... Tactic n.` We remind the reader of the semantics of the tactical THEN: `Tactic 1; Tactic 2` applies Tactic 1 to the current goal, and then Tactic 2 to all the resulting subgoals.

The system allows selecting an arbitrary goal in the list of pending goals, by prefixing the command line by the number of the goal, followed by a colon. This facility is not used in this example, where the current goal is always the first of the list.

3 The proof

3.1 Booleans

3.1.1 $bool = \{true, false\}$, $|bool| = 2$.

The first line is the definition of booleans, as done in the prelude of the system:

```
Inductive Set bool = true : bool | false : bool.
```

Such an inductive definition defines several notions. First it defines the type corresponding to the inductively defined set `bool`, with its constructors `true` and `false`. Next it defines an induction principle `bool_ind`, of type `(P:bool->Prop)(P true)->(P false)->(b:bool)(P b)`, and a recursion principle `bool_rec`, of type `(P:bool->Set)(P true)->(P false)->(b:bool)(P b)`.

Pattern-matching by cases on the constructors of an inductive type T with n constructors, of types respectively $T_1 \rightarrow T$, ... , $T_n \rightarrow T$, may be written with the syntax `(<T'>Match e with e1 ... en)` where e is an expression of type T, and each e_i is of type respectively $T_i \rightarrow T'$.

For instance, here is the definition of a bool predicate `Is_true`, such that `(Is_true true)` is definitionally equal to the true proposition `True`, whereas `(Is_true false)` is definitionally equal to the false proposition `False`:

```
Definition Is_true = [b:bool](<Prop>Match b with
    (* true  *)  True
    (* false *)  False).
```

We show how to prove that $true = false$ is contradictory. We remind the reader of the syntax `<T>A=B` to state that elements A and B, of (set) type T, are equal.

```
Goal ~<bool>true=false.
Unfold not; Intro contr; Change (Is_true false).
```

At this point, the system prints the goal and local hypotheses as follows:

```
1 subgoal
  (Is_true false)
  ============================
    contr : <bool>true=false
```

The elimination tactic applied to hypothesis `contr` will substitute `true` for `false` in the goal. This is because equality is itself defined as an inductive predicate, and its associated induction principle amounts to Leibniz equality. By mere simplification the proof is completed:

```
Elim contr; Simpl; Trivial.
Save diff_true_false.
```

The last command `Save` enters the lemma `diff_true_false` into the global context, with type the original goal statement. The proof above is typical of proofs of "no confusion" properties. If A and B are two terms that we want to prove different, we define a characteristic predicate P, such that $P(A)$ and $\sim P(B)$, we assume the confusion, i.e. $A = B$, convert contradiction into $P(B)$, use $A = B$ to convert it into $P(A)$, trivially true. Clearly, a general tactic could do such proofs in a uniform manner.

3.1.2 Negation

We now define the negation function, which maps `true` to `false` and conversely:

```
Definition neg = [b:bool](<bool>Match b with
    (* true *)  false
    (* false*)  true).
```

We proceed by proving a few easy properties of negation.

```
Goal (b:bool)<bool>(neg (neg b))=b.
Induction b; Simpl; Auto.
Save neg_intro.

Goal (b:bool)<bool>b=(neg (neg b)).
Induction b; Simpl; Trivial.
Save neg_elim.

Goal (b,b':bool)(<bool>b'=(neg b))->(<bool>b=(neg b')).
```

```
Induction b; Induction b'; Intros; Simpl; Trivial.
Save neg_sym.

Goal (b:bool)~<bool>(neg b)=b.
Induction b; Simpl; Unfold not; Intro; Apply diff_true_false; Auto.
Save no_fixpoint_neg.
```

3.2 Boolean words

3.2.1 Strings of bits

We now axiomatize boolean words. This set type is isomorphic to (list bool).

```
Inductive Set word = empty : word
                   | bit : bool -> word -> word.
```

Next is the characteristic predicate of nonempty words:

```
Definition Nonempty = [w:word](<Prop>Match w with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop]True).
```

Notice that here, since word is a genuinely recursive type definition, we get an extra argument P in the bit case, which corresponds to a recursive call to $Nonempty(w)$. Thus `Match` has really the full power of a recursion operateur.

3.2.2 Concatenation

We now define word concatenation. First, we give a "logical" definition, as an inductive ternary relation, in the style of Prolog:

```
Inductive Definition conc : word -> word -> word -> Prop =
  conc_empty : (v:word)(conc empty v v)
| conc_bit   : (u,v,w:word)(b:bool)(conc u v w)
                              ->(conc (bit b u) v (bit b w)).
```

Here is now word concatenation, as a functional definition, in the style of primitive recursion:

```
Definition Conc = [u,v:word](<word>Match u with
    (* empty *)   v
    (* bit b w *) [b:bool][w:word][conc_w_v:word](bit b conc_w_v)).
```

We may now relate the two definitions:

```
Goal (u,v,w:word)(conc u v w)-><word>w=(Conc u v).
Induction 1; Simpl; Trivial.
(*  (u,v,w:word)(b:bool)(conc u v w)->
       (<word>w=(Conc u v))->(<word>(bit b w)=(bit b (Conc u v))) *)
Induction 2; Trivial.
Save conc_Conc.
```

As an exercise, let us give a proof of the associativity of Conc. This property is not used below.

```
Goal (u,v,w:word)<word>(Conc u (Conc v w))=(Conc (Conc u v) w).
Induction u; Simpl; Intros; Auto.
(*  <word>(bit b (Conc y (Conc v w)))=(bit b (Conc (Conc y v) w))
  ============================
    w : word
    v : word
    H : (v:word)(w:word)(<word>(Conc y (Conc v w))=(Conc (Conc y v) w)) *)
Elim (H v w); Trivial.
Save assoc_Conc.
```

Clearly here, we see that it would be easy to increase our proof automation facilities, and get automatically such a proof from a simple indication "By induction on u." We may hope to progressively bridge the gap between our tactic control language, and a genuine mathematical vernacular expressing proof developments.

3.2.3 Singleton words

We end this section with the definition of singleton words, composed of just one bit.

```
Definition single = [b:bool](bit b empty).
```

3.3 Alternating words

A word w is *alternate* if for some bit b, w is of the form `[`$b\ \sim b\ b\ \sim b...$`]`. We write `(alt b w)`.

3.3.1 Alt and its variants

```
Inductive Definition alt : bool -> word -> Prop =
  alt_empty   : (b:bool)(alt b empty)
| alt_bit     : (b:bool)(w:word)(alt (neg b) w)->(alt b (bit b w)).

Hint alt_empty alt_bit.
```

This `Hint` command is a pragma for the automated search tactics `Trivial` and `Auto`. It tells them to apply the clause `alt_bit` whenever possible, in order to simplify a goal of the form `(alt b (bit b w))` into `(alt (neg b) w)`. We now proceed with a more functional characterisation of alternate.

```
Definition Alt = [b:bool][w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b':bool][w:word][P:Prop](alt (neg b) w)).
```

Same as previously, we want to "invert" our definition clauses. Here we have an extra difficulty, due to the non-linearity of the head of clause `alt_bit`: the variable b occurs twice in $(alt\ b\ (bit\ b\ w))$.

```
Goal (b,b':bool)(w:word)(alt b (bit b' w))->(alt (neg b) w).
Intros b b' w al.
Change (Alt b (bit b' w)).
Elim al; Simpl; Trivial.
Save alt_neg_intro.

Goal (b,b':bool)(w:word)(alt (neg b) (bit b' w))->(alt b w).
Intros; Elim (neg_intro b); Apply alt_neg_intro with b'; Trivial.
Save alt_neg_elim.
```

We now take care of the linearity condition. What we have done is to break the property into two: $(alt\ b\ w) \leftrightarrow (Alt\ b\ w) \wedge (Alt'\ b\ w)$.

```
Definition Alt' = [b:bool][w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b':bool][w:word][P:Prop]<bool>b=b').

Goal (b,b':bool)(w:word)(alt b (bit b' w))-><bool>b=b'.
Intros b b' w al.
Change (Alt' b (bit b' w)).
Elim al; Simpl; Trivial.
Save alt_eq.

Goal (b,b':bool)(w:word)(alt b (bit b' w))->((<bool>b=b') /\ (alt (neg b) w)).
Intros; Split.
Apply alt_eq with w; Trivial.
Apply alt_neg_intro with b'; Trivial.
Save alt_back.
```

We end with the existence property: w is *alternate* if for some bool b we have $(alt\ b\ w)$. We use directly an inductive definition, rather than an explicit existential quantification, which is itself defined inductively anyway.

```
Inductive Definition alternate [w:word] : Prop =
  alter : (b:bool)(alt b w)->(alternate w).
```

3.4 Parities of words

We give here definitions of *odd* and *even* words. This is the corresponding property of their length, but we do this directly, without resorting to definition of length, and arithmetic facts. As previously, *Odd* (resp. *Even*) is the functional notion corresponding to the logical one *odd* (resp. *even*).

3.4.1 Odd and Even

```
Inductive Definition odd : word -> Prop =
  odd_single : (b:bool)(odd (single b))
| odd_bit    : (w:word)(odd w)->(b,b':bool)(odd (bit b (bit b' w))).
```

```
Definition Odd = [w:word](<Prop>Match w with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match w with
         (* empty *)   True
         (* bit b w *) [b:bool][w:word][P:Prop](odd w))).

Inductive Definition even : word -> Prop =
  even_empty : (even empty)
| even_bit : (w:word)(even w)->(b,b':bool)(even (bit b (bit b' w))).

Hint even_empty.

Definition Even = [w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b:bool][w:word][P:Prop](<Prop>Match w with
         (* empty *)   False
         (* bit b w *) [b:bool][w:word][P:Prop](even w))).
```

3.4.2 Parity lemmas

```
Goal ~(odd empty).
Unfold not; Intro od.
Change (Nonempty empty).
Elim od; Simpl; Trivial.
Save not_odd_empty.

Hint not_odd_empty.

Goal (w:word)(b,b':bool)(odd (bit b (bit b' w)))->(odd w).
Intros w b b' od.
Change (Odd (bit b (bit b' w))).
Elim od; Simpl; Trivial.
Save odd_down.

Definition Not_single = [w:word](<Prop>Match w with
    (* empty *)   True
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match w with
         (* empty *)   False
         (* bit b w *) [b:bool][w:word][P:Prop]True)).

Goal (b:bool)~(even (single b)).
Intro b; Unfold not; Intro ev.
Change (Not_single (single b)).
Elim ev; Simpl; Trivial.
Save not_even_single.
```

```
Goal (w:word)(b,b':bool)(even (bit b (bit b' w)))->(even w).
Intros w b b' ev.
Change (Even (bit b (bit b' w))).
Elim ev; Simpl; Trivial.
Save even_down.

Goal (w:word)(odd w)->(b:bool)(even (bit b w)).
Induction 1.
Intros b b'; Unfold single; Apply even_bit; Apply even_empty.
Intros; Apply even_bit; Auto.
Save odd_even.

Goal (w:word)(even w)->(b:bool)(odd (bit b w)).
Induction 1.
Intro b; Change (odd (single b)); Apply odd_single.
Intros; Apply odd_bit; Auto.
Save even_odd.

Hint odd_even even_odd.

Goal (w:word)(b:bool)(odd (bit b w))->(even w).
Induction w; Auto.
Intros; Apply odd_even; Apply odd_down with b0 b; Auto.
Save inv_odd.

Goal (w:word)(b:bool)(even (bit b w))->(odd w).
Induction w; Intros.
Absurd (even (single b)); Trivial.
Apply not_even_single.
Apply even_odd.
Apply even_down with b0 b; Trivial.
Save inv_even.
```

3.4.3 (odd w) $\oplus$ (even w)

In this section, we prove that *odd* and *even* are two disjoint complementary properties.

```
Goal (w:word)((odd w) \/ (even w)).
Induction w; Auto.
Induction 1; Intros.
Right; Auto.
Left; Auto.
Save odd_or_even.

Goal (w:word)(odd w)->(even w)->False.
Induction w; Intros.
Elim not_odd_empty; Trivial.
```

```
(*   False
  ============================
    H1 : (even (bit b y))
    H0 : (odd (bit b y))
    H : (odd y)->(even y)->False *)
Apply H.
Apply inv_even with b; Trivial.
Apply inv_odd with b; Trivial.
Save not_odd_and_even.
```

3.4.4 Parities of subwords

When we cut an even word w into subwords u and v, they are both even or both odd. But if w is odd, one is odd and the other even. This easy consequence of `odd_or_even` could be automated with a simple propositional tactic.

```
Goal (u,v,w:word)(conc u v w)->
           (((odd w) /\
                   ( ((odd u) /\ (even v))
                  \/ ((even u) /\ (odd v))))
        \/ ((even w) /\
                   ( ((odd u) /\ (odd v))
                  \/ ((even u) /\ (even v))))).
Induction 1; Intros.
Elim (odd_or_even v0); Intro.
Left; Split; Auto.
Right; Split; Auto.
Elim H1; Intros.
(* 1 (odd w0) *)
Right; Elim H2; Intros.
Split; Auto.
Elim H4; Intros.
Right; Split; Elim H5; Auto.
Left; Split; Elim H5; Auto.
(* 2 (even w0) *)
Left; Elim H2; Intros.
Split; Auto.
Elim H4; Intros.
Right; Split; Elim H5; Auto.
Left; Split; Elim H5; Auto.
Save odd_even_conc.
```

We are specially interested in the even case, which we explicitate below. This is typical: it is often simpler to prove a general disjunctive statement by induction, and then to particularise it using the fact that some cases are disjoint.

```
Goal (u,v,w:word)(conc u v w)->(even w)->
                   ( ((odd u) /\ (odd v))
```

```
                  \/ ((even u) /\ (even v))).
Intros u v w c e; Elim odd_even_conc with u v w; Intros.
Elim H; Intro o; Elim not_odd_and_even with w; Auto.
Elim H; Intros; Trivial.
Trivial.
Save even_conc.
```

3.4.5 Subwords of alternate words are alternate

```
Goal (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->(alt b u).
Induction 1; Auto; Intros.
(*  (alt b0 (bit b u0)) *)
Elim alt_back with b0 b w0.
Intros eq A.
(*   (alt b0 (bit b u0))
  ============================
    A : (alt (neg b0) w0)
    eq : <bool>b0=b *)
Elim eq; Auto.
Trivial.
Save alt_conc_l.
```

This was simple enough for the prefix part. For the suffix part, we have a parity consideration:

```
Goal (u,v,w:word)(conc u v w)->(b:bool)(alt b w)->
   ( ((odd u) /\ (alt (neg b) v))
  \/ ((even u) /\ (alt b v))).
Induction 1; Intros.
Right; Split; Intros; Auto.
(*  ((odd (bit b u0)) /\ (alt (neg b0) v0))
  \/ (even (bit b u0)) /\ (alt b0 v0)
  ============================
    H2 : (alt b0 (bit b w0))
    b0 : bool
    H1 : (b:bool)(alt b w0)->
          (((odd u0) /\ (alt (neg b) v0)) \/ (even u0) /\ (alt b v0)) *)
Elim H1 with (neg b0).
Elim neg_elim with b0; Intro.
Right; Split; Elim H3; Auto.
Intro; Left; Split; Elim H3; Auto.
Apply alt_neg_intro with b; Trivial.
Save alt_conc_r.
```

Putting the two together:

```
Goal (u,v,w:word)(conc u v w)->(alternate w)->((alternate u) /\ (alternate v)).
Induction 2; Intros b ab; Split.
```

```
Apply alter with b; Apply alt_conc_l with v w; Auto.
Elim alt_conc_r with u v w b; Intros; Trivial.
Elim H1; Intros; Apply alter with (neg b); Auto.
Elim H1; Intros; Apply alter with b; Auto.
Save alt_conc.
```

This last lemma is actually not used below.

3.5 Opposite words

Two words are said to be *opposite* if they start with different bits:

```
Inductive Definition opposite : word -> word -> Prop =
  opp : (u,v:word)(b:bool)(opposite (bit b u) (bit (neg b) v)).

Definition Opp = [u:word][v:word](<Prop>Match u with
    (* empty *)   False
    (* bit b w *) [b:bool][w:word][P:Prop](
      <Prop>Match v with
         (* empty *)   False
         (* bit b w *) [b':bool][w':word][P':Prop]<bool>(neg b)=b')).

Goal (u:word)~(opposite u empty).
Unfold not; Intros u op.
Change (Nonempty empty).
Elim op; Simpl; Trivial.
Save not_opp_empty_r.

Goal (u:word)~(opposite empty u).
Unfold not; Intros u op.
Change (Nonempty empty).
Elim op; Simpl; Trivial.
Save not_opp_empty_l.

Goal (u,v:word)(b:bool)~(opposite (bit b u) (bit b v)).
Unfold not; Intros u v b op.
Apply (no_fixpoint_neg b).
Change (Opp (bit b u) (bit b v)).
Elim op; Simpl; Trivial.
Save not_opp_same.

Goal (u,v:word)(b:bool)(odd u)->(alt b u)->
               (odd v)->(alt (neg b) v)->(opposite u v).
Induction u.
Intros v b odd_empty;
Absurd (odd empty); Trivial.
Intros b u' H v; Elim v.
Intros b' H1 H2 odd_empty.
```

```
Absurd (odd empty); Trivial.
Intros b' v' H' b'' H1 H2 H3 H4.
(*  (opposite (bit b u) (bit b' v'))
  ============================
    H4 : (alt (neg b'') (bit b' v'))
    H3 : (odd (bit b' v'))
    H2 : (alt b'' (bit b u')) *)
Elim (alt_eq (neg b'') b' v'); Trivial.
Elim (alt_eq b'' b u'); Trivial.
Apply opp.
Save opposite1.

Goal (u,v:word)(b:bool)(alt b u)->(alt b v)->~(opposite u v).
Induction u.
Intros; Apply not_opp_empty_l.
Intros b u' H v; Elim v.
Intros; Apply not_opp_empty_r.
Intros b' v' H1 b'' H2 H3.
(*   ~(opposite (bit b u') (bit b' v'))
  ============================
    H3 : (alt b'' (bit b' v'))
    H2 : (alt b'' (bit b u')) *)
Elim (alt_eq b'' b' v' H3).
Elim (alt_eq b'' b u' H2).
Apply not_opp_same.
Save opposite2.
```

3.6 Paired words

A word w is said to be *paired* if it is of the form: [$b_1 \ \sim b_1 \ b_2 \ \sim b_2 \ ... \ b_n \ \sim b_n$].

```
Inductive Definition paired : word -> Prop =
  paired_empty : (paired empty)
| paired_bit : (w:word)(paired w)->(b:bool)(paired (bit (neg b) (bit b w))).
```

Here again we have a non-linear clause. We shall give several variants of "paired". A paired word must be even. For odd words, we give below variants paired_odd_l and paired_odd_r.

$$(paired\_odd\_l\ b\ w) \leftrightarrow w = [b\ b_1\ \sim b_1\ b_2\ \sim b_2\ ...\ b_n\ \sim b_n].$$

```
Definition paired_odd_l = [b:bool][w:word](paired (bit (neg b) w)).

Goal (b:bool)(w:word)(paired w)->(paired_odd_l b (bit b w)).
Unfold paired_odd_l; Intros.
Apply paired_bit; Trivial.
Save paired_odd_l_intro.

Goal (b:bool)(w:word)(paired_odd_l (neg b) w)->(paired (bit b w)).
```

```
Unfold paired_odd_l; Intros.
Elim (neg_intro b); Trivial.
Save paired_odd_l_elim.
```

Similarly:

$$(paired\_odd\_r\ b\ w) \leftrightarrow w = [b_1\ \sim b_1\ b_2\ \sim b_2\ ...\ b_n\ \sim b_n\ \sim b].$$

```
Definition paired_odd_r = [b:bool][w:word](paired (Conc w (single b))).
```

An even word is *paired rotated* iff rotating it by one bit makes it paired:

$$(paired\_rot\ b\ w) \leftrightarrow w = [b\ b2\ \sim b2\ ...\ bn\ \sim bn\ \sim b].$$

```
Inductive Definition paired_rot : bool -> word -> Prop =
  paired_rot_empty : (b:bool)(paired_rot b empty)
| paired_rot_bit : (b:bool)(w:word)(paired_odd_r b w)
                                ->(paired_rot b (bit b w)).

Goal (w:word)(b:bool)(paired_rot b w)->(paired_odd_r b (bit (neg b) w)).
Induction 1.
Intro; Unfold paired_odd_r; Simpl.
Unfold single; Apply paired_bit.
Apply paired_empty.
Intros b0 b' w'; Unfold paired_odd_r; Intros.
Simpl; Apply paired_bit; Auto.
Save paired_odd_r_from_rot.
```

Finally, a word is said to be *paired between* if it is obtained by prefixing and suffixing a paired word with the same bit b:

$$(paired\_bet\ b\ w) \leftrightarrow w = [b\ b_1\ \sim b_1\ b_2\ \sim b_2\ ...\ b_n\ \sim b_n\ b].$$

```
Inductive Definition paired_bet [b:bool] :  word -> Prop =
  paired_bet_bit : (w:word)(paired_odd_r (neg b) w)->(paired_bet b (bit b w)).

Goal (b:bool)(w:word)(paired_bet (neg b) w)->(paired_odd_r b (bit b w)).
Intros b w pb.
Elim (neg_intro b).
Elim pb.
Unfold * paired_odd_r. (* Unfold twice *)
Intros; Simpl.
Apply paired_bit; Trivial.
Save paired_odd_r_from_bet.
```

3.7 Shuffling two words

Here we come to our main notion: `(shuffle u v w)`, meaning word w may be obtained by shuffling words u and v. We deal here with a truly non-deterministic specification.

3.7.1 Random shuffle

```
Inductive Definition shuffle : word -> word -> word -> Prop =
  shuffle_empty :    (shuffle empty empty empty)
| shuffle_bit_left : (u,v,w:word)(shuffle u v w) ->
                            (b:bool)(shuffle (bit b u) v (bit b w))
| shuffle_bit_right : (u,v,w:word)(shuffle u v w) ->
                            (b:bool)(shuffle u (bit b v) (bit b w)).
```

3.7.2 The shuffling lemma

This lemma is the main result of this note. It gives the inductive invariant associated with the shuffling of alternated words.

```
Goal (u,v,w:word)(shuffle u v w)->(b:bool)(alt b u)->
  ( ((odd u) /\ ( ((odd v)  /\ ((alt (neg b) v) -> (paired w))
                            /\ ((alt b v)       -> (paired_bet b w)))
              \/ ((even v) /\ ((alt b v)       -> (paired_odd_l b w)))
                            /\ ((alt (neg b) v) -> (paired_odd_r (neg b) w))))
 \/ ((even u) /\ ( ((odd v) /\ ((alt (neg b) v) -> (paired_odd_r b w))
                            /\ ((alt b v)       -> (paired_odd_l b w)))
              \/ ((even v) /\ ((alt b v)       -> (paired_rot b w))
                            /\ ((alt (neg b) v) -> (paired w)))))).
Induction 1; Intros.
(* 0. empty case *)
Right.
Split; Auto.
Right.
Split; Auto.
Split; Intro.
Apply paired_rot_empty.
Apply paired_empty.
(* 1. u before v *)
Elim (alt_eq b0 b u0); Trivial.
Elim (H1 (neg b0)); Intros.
(* 1.1 *) Right.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
Elim H6; Intros.
(* 1.1.1 *) Left.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_r_from_bet; Auto.
Apply paired_odd_l_intro; Apply H9; Elim (neg_elim b0); Auto.
Elim H6;Intros.
Elim H7; Intros.
```

```
(* 1.1.2 *) Right.
Split; Auto.
Split; Intro.
Apply paired_rot_bit; Elim (neg_intro b0); Apply H8; Elim (neg_elim b0); Auto.
Apply paired_odd_l_elim; Auto.
(* 1.2 *) Left.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 1.2.1 *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intro.
Apply paired_odd_l_elim; Auto.
Apply paired_bet_bit; Apply H9; Elim (neg_elim b0); Auto.
(* 1.2.2 *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split.
Split; Auto.
Intro.
Apply paired_odd_l_intro; Apply H10; Elim (neg_elim b0); Auto.
Intro; Pattern 2 b0; Elim (neg_intro b0).
Apply paired_odd_r_from_rot; Auto.
Apply alt_neg_intro with b; Trivial.
(* 2. v before u *)
Elim (H1 b0); Intros.
(* 2.1 *) Left.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 2.1.1 *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split.
Split; Auto.
Intro.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_odd_l_intro; Apply H9; Apply alt_neg_intro with b; Auto.
Intro.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_r_from_bet; Elim (neg_elim b0); Apply H10;
 Apply alt_neg_elim with b; Auto.
(* 2.1.2 *) Left.
Elim H6; Intros.
Elim H7; Intros.
```

```
Split; Auto.
Split; Intros.
Apply paired_odd_l_elim.
Elim (alt_eq (neg b0) b v0); Trivial.
Elim (neg_elim b0).
Apply H10.
Elim (neg_intro b0).
Apply alt_neg_intro with b; Auto.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_bet_bit; Apply H8; Apply alt_neg_intro with b; Auto.
(* 2.2 *) Right.
Elim H3; Intros.
Split; Auto.
Elim H5; Intros.
(* 2.2.1 *) Right.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intros.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_rot_bit; Apply H9; Apply alt_neg_intro with b; Auto.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_l_elim.
Elim (neg_elim b0); Apply H10; Elim (neg_intro b0);
 Apply alt_neg_intro with b; Auto.
(* 2.2.2 *) Left.
Elim H6; Intros.
Elim H8; Intros.
Split; Auto.
Split; Intros.
Elim (alt_eq (neg b0) b v0); Trivial.
Apply paired_odd_r_from_rot; Apply H9; Elim (neg_intro b0);
 Apply alt_neg_intro with b; Auto.
Elim (alt_eq b0 b v0); Trivial.
Apply paired_odd_l_intro; Apply H10; Apply alt_neg_intro with b; Auto.
Trivial.
Save Shuffling.
```

Remark. This proof has the natural propositional structure to examine the eight cases. There are thus eight trivial cases to examine in the base case (shuffling two empty lists), and eight sub-cases in each of the two non-degenerate cases of `shuffle_bit`. We thus appeal to sixteen sub-lemmas, corresponding to "verification conditions" in Floyd-Hoare terminology, the "program" being "evidence" that the result of the shuffle of subwords of an even alternated word is paired. These sixteen verification conditions reduce to the six lemmas: `paired_odd_l_intro` and `paired_odd_l_elim` (each used four times), `paired_rot_bit`, `paired_bet_bit`, `paired_odd_r_from_bet`, and `paired_odd_r_from_rot` (each used twice). Now the reader may wonder by what stroke of luck we just developed the right lemmas above.

Of course the development of the proof went top-down, not only subgoaling locally in each lemma, but globally as well. I.e., the statements of the above six lemmas were precisely generated by the proof of `Shuffling`. This is extremely easy in our system, where we may at any point of the proof declare a new axiom, which gets inserted in the global context and is thus usable instantly. A second pass changes these "axioms" into goals themselves to be proven.

3.8 The main theorem

3.8.1 Rotating a word

We express the `paired_rot` property as the composition of a rotating operation `rotate` and of the property `paired`.

$$rotate\ [b_1\ b_2\ ...\ b_n] = [b_2\ ...\ b_n\ b_1].$$

```
Definition rotate [w:word](Elim{w,word}
   (* empty *)    empty
   (* bit b u *)  [b:bool][u,v:word](Conc u (single b))).

Goal (w:word)(b:bool)(paired_rot b w)->(paired (rotate w)).
Induction 1.
Intro; Simpl; Apply paired_empty.
Intros b' w'; Simpl.
Unfold paired_odd_r; Trivial.
Save paired_rotate.
```

We now prove the main theorem of this note. We use the section mechanism in order to state our hypotheses in the global context and shorten the quantification prefixes.

3.8.2 Stating the hypotheses of the main result

```
Section Main.
Variable x:word.
Hypothesis Even_x : (even x).
Variable b:bool. (* witness for (alternate x) *)
Hypothesis A : (alt b x).
Variables u,v:word.
Hypothesis C : (conc u v x).
Variable w:word.
Hypothesis S : (shuffle u v w).

Goal (alt b u).
Apply alt_conc_l with v x.
Apply C.
Apply A.
Save Alt_u.
```

3.8.3 Case 1: u is odd

```
Section Case1.
```

```
Hypothesis Odd_u : (odd u).

Goal ~(even u).
Red; Intro.
Elim not_odd_and_even with u; Trivial.
Apply Odd_u.
Save Not_even_u.

Goal (odd v).
Elim even_conc with u v x.
Intro H; Elim H; Trivial.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply C.
Apply Even_x.
Save Odd_v.

Goal (alt (neg b) v).
Elim alt_conc_r with u v x b.
Intro H; Elim H; Trivial.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply C.
Apply A.
Save Alt_v.

Goal (opposite u v).
Apply opposite1 with b.
Apply Odd_u.
Apply Alt_u.
Apply Odd_v.
Apply Alt_v.
Save Opp_uv.

Goal (paired w).
Elim Shuffling with u v w b.
Intro H; Elim H; Intros H1 H2; Elim H2; Intros.
Elim H0; Intros.
Elim H4; Intros.
Apply H5.
Apply Alt_v.
Intros; Elim H0; Intros.
Elim H3; Intros.
Elim not_odd_and_even with v; Trivial.
Apply Odd_v.
Intro H; Elim H; Intro; Elim Not_even_u; Trivial.
Apply S.
Apply Alt_u.
Save Case1.
```

```
End Case1.
```

3.8.4 Case 2: u is even

```
Section Case2.
Hypothesis Even_u : (even u).

Goal ~(odd u).
Red; Intro; Elim not_odd_and_even with u; Trivial.
Apply Even_u.
Save Not_odd_u.

Goal (even v).
Elim even_conc with u v x.
Intro H; Elim H; Intro; Elim Not_odd_u; Trivial.
Intro H; Elim H; Trivial.
Apply C.
Apply Even_x.
Save Even_v.

Goal (alt b v).
Elim alt_conc_r with u v x b.
Intro H; Elim H; Intro; Elim Not_odd_u; Trivial.
Intro H; Elim H; Trivial.
Apply C.
Apply A.
Save Alt_v.

Goal ~(opposite u v).
Apply opposite2 with b.
Apply Alt_u.
Apply Alt_v.
Save Not_opp_uv.

Goal (paired (rotate w)).
Apply paired_rotate with b.
Elim Shuffling with u v w b.
Intro case_odd_u; Elim case_odd_u; Intro; Elim Not_odd_u; Trivial.
Intro case_even_u; Elim case_even_u; Intros even_u odd_even_v; Elim odd_even_v.
Intro odd_v; Elim odd_v; Intros; Elim not_odd_and_even with v; Trivial.
Apply Even_v.
Intro case_even_v; Elim case_even_v; Intros even_v paired_hyp.
Elim paired_hyp; Intros case_b case_neg_b.
Apply case_b.
Apply Alt_v.
Apply S.
Apply Alt_u.
```

```
Save Case2.

End Case2.
```

3.8.5 Putting it all together

We recall the conditional connective, from the prelude, with its mixfix syntax:

```
Definition IF = [P,Q,R:Prop](P /\ Q) \/ ((~P) /\ R).
Syntax IF "if _ then _ else _".

Goal if (opposite u v) then (paired w) else (paired (rotate w)).
Unfold IF; Elim odd_or_even with u; Intros.
(* (odd u) : Case 1 *)
Left; Split.
Apply Opp_uv; Trivial.
Apply Case1; Trivial.
(* (even u) : Case 2 *)
Right; Split.
Apply Not_opp_uv; Trivial.
Apply Case2; Trivial.
Save Main.

End Main.
```

At this point all the local hypotheses are discharged into a general Main theorem explicitly universally quantified.

3.9 Gilbreath's trick

All is left to do is to hide the existential introduction of hypothesis b above, using the `alternate` property:

```
Goal (x:word)(even x)->(alternate x)->(u,v:word)(conc u v x)->
     (w:word)(shuffle u v w)->
     if (opposite u v) then (paired w) else (paired (rotate w)).

Induction 2; Intros. (* Existential intro *)
Apply Main with x b; Trivial.
Save Gilbreath_trick.
```

This Concludes our proof of Gilbreath's card trick.

4 The trick

Why is this a card trick? Our boolean words are card decks, with true for red and false for black. Take an even deck x, arranged alternatively red, black, red, black, etc. Ask a spectator to cut the deck, into sub-decks u and v. Now shuffle u and v into a new deck w. When shuffling, note carefully whether u and v start with opposite colors or not. If they do, the resulting deck is composed of pairs red-black or black-red; otherwise, you get the property by first rotating the deck by one card. The trick is usually played by putting the deck behind your back after the shuffle, to perform "magic". The magic is either rotating or doing nothing. When showing the pairing property, say loudly "red black red black..." in order to confuse in the spectator's mind the weak *paired* property with the strong *alternate* one.

There is a variant. If the cut is favorable, that is if u and v are opposite, just go ahead showing the pairing, without the "magic part." If the spectator says that he understands the trick, show him the counter-example in the non-favorable case. Of course now you have to leave him puzzled, and refuse to redo the trick.

References

[1] R. Boyer, J Moore. "A Computational Logic." Academic Press (1979).

[2] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[3] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[4] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[5] N.G. de Bruijn. "A riffle shuffle card trick and its relation to quasi crystal theory." Nieuw Archief Voor Wiskunde **5** 3 (1987) 285–301.

[6] R.L. Constable et al. "Implementing Mathematics in the Nuprl System." Prentice-Hall (1986).

[7] R.L. Constable and N.P. Mendler. "Recursive Definitions in Type Theory." In Proc. Logic of Programs, Springer-Verlag Lecture Notes in Computer Science **193** (1985).

[8] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[9] T. Coquand. "Metamathematical Investigations of a Calculus of Constructions." Rapport de recherche INRIA 1088, Sept. 89. In "Logic and Computer Science," ed. P. Odifreddi, Academic Press, 1990, 91–122.

[10] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[11] T. Coquand and C. Paulin-Mohring. "Inductively defined types." Workshop on Programming Logic, Göteborg University, Båstad, (89). International Conference on Computer Logic COLOG-88, Tallinn, Dec. 1988. LNCS 417, P. Martin-Löf and G. Mints eds., pp. 50-66.

[12] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, B. Werner. "The system Coq, V5.6, User's guide." INRIA Technical Report, to appear, Sept. 1991.

[13] M. Gardner. Mathematical Recreation column, Scientific American, Aug. 1960.

[14] M. Gardner. Chapter 9, "New Mathematical Diversions from Scientific American." George Allen and Unwin Ltd, London, 1966. Reprinted Simon and Schuster, 1971.

[15] N. Gilbreath. "Magnetic Colors." The Linking Ring, **38** 5 (1959).

[16] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF." Springer-Verlag LNCS **78** (1979).

Completing the Rationals and Metric Spaces in LEGO*

Claire Jones[†]
LFCS,
University of Edinburgh

June 1991

Abstract

This paper will describe the work carried out in implementing (or proving) some elementary theorems of constructive analysis in the Extended Calculus of Constructions using the LEGO system. Two main areas have been considered, firstly the construction of the reals from the rationals and secondly the completion of a metric space (defining a constructions function of type `Metric -> Metric`). Many interesting issues arise from this e.g. how best to represent sets, subsets and functions.

1 Introduction and Related Work

In this paper we will describe the research at Edinburgh where we have been implementing some theorems of analysis on the Lego system of Randy Pollack [8]. We have been using this system to work within the Calculus of Constructions of Huet and Coquand [4], and the extension of this with sigma types and type universes due to Luo [7].

This work has progressed much with the aid of Gabe Stolzenberg whose formulation of constructive mathematics we have been following quite closely. In choosing to implement constructive mathematics we have departed from the path taken by De Bruijn and the Automath project (see [5]).

We have proved completely one theorem: the completion of a metric space and have done the major part of the construction of the reals from the rationals. To accomplish this we had to code up the basic mathematical ideas of sets, functions, relations and subsets in the Calculus of Constructions. The last is known to be particularly problematic in type theories. Other problems we had to solve included finding or making a coherent and reasonably detailed account of the theorems we wished to prove—and perhaps more importantly of the necessary constructions and definitions used in the proof. In fact we never really solved this problem for the reals, there were just too many functions and axioms about them and too many simple but necessary lemmas. We had to evolve strategies for naming large collections of lemmas and for splitting proofs which were large for the system in their natural formulation. We had to discover how to express lemmas so that they were easy to use with the system.

This work follows on from that done by the NuPrl project at Cornell [3] where they have implemented some constructive analysis mainly taken from Bishop [2].

Related work includes the Coq system at INRIA [6] and in Gothenburg the ALF system [1].

*Supported by the SERC with contribution from the BRA Logical Frameworks

[†] `ccmj@dcs.ed.ac.uk`

2 The Theorems

In this section we will give an outline of the proofs of the two theorems we have been working on. Both the theorems involve constructing completions of mathematical objects with respect to a distance function, so we will first define what completeness means and then go on to the theorems.

2.1 Completeness

If we have some space with a distance function (a metric in the case of a metric space, and subtraction in the case of the real numbers or the rationals) which satisfies some simple axioms we can define the notion of *completeness*. Informally a space is called "complete" if there are no "holes" in it. In classical mathematics we talk about a sequence of points $\{x_n\}$, for n ranging over the natural numbers, converging to a limit x (which is unique if it exists) if for all $\epsilon > 0$ we have an N such that $d(x_m, x) < \epsilon$ for all m greater then N. A sequence is called *Cauchy* if the points in it get closer together in a certain way: formally if for all $\epsilon > 0$ there is an N s.t. $d(x_m, x_n) < \epsilon$ whenever both m and n are greater than N. A space is complete if all Cauchy sequences have a limit.

In constructive mathematics we will not use the notion of a Cauchy sequence but instead use an equivalent notion which carries around more explicitly the Cauchy property in that each element of the "sequence" has a length associated with it which says how far away the element is from the limit. In the construction of the reals from the rationals we use collections of intervals, where the limit of a collection of intervals is their intersection. In the metric space theorem we use balls, a point and a radius, with the limit again being the intersection. Naturally not every collection of intervals or balls will have a limit: in our theorems we will state the necessary conditions for a limit to exist. In each case the limit can be defined as a function.

There will be no mention of topology in these proofs. The reason for this is that in constructive analysis the notion of a topological space is found to be uninteresting (see [2]). So notions like the limit of a convergent sequence are defined on metric spaces in terms of a distance function.

2.2 Completing the Rationals

If we complete the rationals, it is well-known that the result is what is known as the real numbers. Classically the reals are the only complete, ordered field. (A *field* has two operations, (commutative) addition and multiplication with inverses (except for zero) and with multiplication distributing over addition, examples include the reals, R^n, the complex numbers. An *ordered field* can be defined as a field with a distinguished subset of "positive" elements, closed under multiplication and addition such that every element is either zero, positive or its additive inverse is positive. Equivalently there is a total order such that the order relation relates to addition and multiplication in the obvious way. See e.g. [9] for more details.)

Theorem We construct a complete ordered field containing the rationals on which the operations of multiplication and addition are extended from their usual definition on the rationals.

For further details on this theorem see [11]. I will just give a brief outline. Note that we haven't yet implemented all of the proof in lego, in a later section we will note which parts of the proof have yet to be completed.

We assume we already have defined or constructed the field of the rational numbers, including the operations of multiplication, addition, subtraction and division, and also the order relation (less than and strictly less than). The equality and order are decidable in the field; this stems from the decidability in the natural numbers although in fact we rarely need it in the definition of the reals.

Proof We first define the notion of an *interval* with rational end-points. This interval has a length, namely the upper end-point minus the lower. Given a collection of intervals, we consider two important properties:

Consistency any pair of intervals in the collection overlap.

Arbitrarily Small Length that the collection contains intervals of arbitrarily small length.

An element of the completion (a *real number*) will be a collection of intervals which satisfies these two properties. Intuitively the real number is the number specified by the intersection of these intervals. As an example we may consider the collection of intervals

$$\{[a, b] \mid a^2 \leq 2 \leq b^2 \text{ and } a, b \geq 0\}$$

which defines the real number $\sqrt{2}$ and can be shown to have the above two properties.

We must define an *equality relation* on these collections of intervals, and prove that it is symmetric, transitive and reflexive. This will give us the (undecidable) equality on the real numbers. We say that two real numbers are equal if the union of their collections of intervals is consistent, i.e. any pair of intervals from the two numbers overlap. Symmetry and reflexivity of this relation are trivial to prove. Transitivity is not: we show that any two intervals overlap by showing that the distance between the endpoints of an arbitrary pair of intervals is less than any positive rational number, hence must be zero, so they overlap.

So we have defined the set of real numbers. We note the inclusion of the rationals into this set, given by sending a rational q to the collection of intervals consisting only of the singleton interval $[q, q]$ which can be shown to have the required properties. Later we will note that addition and multiplication on the reals will extend the existing addition and multiplication on the rationals via this inclusion.

We now have to define functions on the set of reals. We can get a categorical theorem saying generally when we can extend functions in the usual way by expressing the completion as a monadic functor. But this theorem is too weak. We can only extend functions which are uniformly continuous on the whole real line, this rules out multiplication, square and reciprocal for a start. One way around this is to consider extending functions which are uniformly continuous on finite intervals; for that we should note that we can complete the rationals in some finite interval and embed the resulting set into the reals in the obvious way. It is still not very easy to find a lemma which will work on functions with different arities. The other way is to treat each function separately; this makes sense for the reals as we really only have the basic arithmetic operations to worry about. Other functions on the reals are generally defined using power-series which we can get from addition and multiplication.

The general method of extending a function from the rationals to the reals is to apply the function to the end-points of the collection of intervals. But in each case we have to be a little careful to get the correct intervals out, particularly multiplication where there are problems with sign changes.

We first define addition and multiplication. Subtraction and division can be handled by defining the unary operation $x \mapsto -x$ and reciprocals. Obviously we have be careful with reciprocal since it is not a total function (not defined unless $x <> 0$). Before we can define reciprocal we thus have to have the $<$ relation, although this is straightforward.

To define addition, we simply define an operation on the collections of intervals, namely

$$I + J = \{[x + a, y + b] \mid [x, y] \in I, [a, b] \in J\}$$

where I and J are two collections of intervals. It is then fairly straightforward to show that the resulting collection has the required properties. For multiplication it is slightly less straightforward essentially because of the behaviour of negative numbers and zero. We must define the product of two intervals $[x, y]$ and $[a, b]$ as $[\min(xa, xb, ya, yb), \max(xa, xb, ya, yb)]$. Again, it is straightforward to show the resulting collection of intervals has the required properties, but the algebra becomes quite involved. Subtraction is straight-forward. For reciprocal, we need to know that the real number is non-zero. We can define this by first defining $I < J$ iff there exists intervals $[a, b] \in I$ and $[c, d] \in J$ with $b < c$. Then $I \leq J$ is defined as (not $I > J$), i.e. for any $[a, b] \in I$ and $[c, d] \in J$, $a \leq d$. Note here that for rationals, (not $a \leq b$) is equivalent to $b < a$, but this does *not* hold for reals. Given these definitions we can define the reciprocal function, for collections of intervals with either $I < 0$ or $I > 0$ where 0 is the injection of the rational number zero into the system of reals. Essentially we just take every interval $[a, b]$ in I which doesn't have zero as an endpoint, and replace it by $[1/b, 1/a]$. The problem comes when we want to find intervals of arbitrarily small length, since $(1/a - 1/b) = (b - a)/ab$ so we need to find a lower bound for ab. This is possible although quite complicated. In fact we have to do a similar thing for multiplication as well, but this time we need an upper bound.

Finally with all these definitions we can state and then prove the completeness theorem for the reals. We state this theorem in terms of intervals with the properties mentioned above, but this time the intervals have real end-points, so we need to define length using subtraction on the reals and we need $\leq$ to define the interval. We define the limit of some appropriate collection of intervals essentially using the big union operation, since we take as intervals, all rational $[a, d]$ where $[r, s]$ is an real interval in the collection and $[a, b]$ is an interval for r and $[c, d]$ is an interval for s. Then the overlap part is trivial. The more difficult part is finding intervals of arbitrarily small length; we need some algebra about subtraction, and how it relates to subtraction in the rationals.

2.3 Completing a Metric Space

A metric space is a set of points with a real-valued distance function between them (usually denoted $d(x, y)$) which satisfies the following axioms:

1. $d(x, y) = 0$ iff $x = y$.
2. $d(x, y) \geq 0$.
3. $d(x, y) = d(y, x)$ (symmetry).
4. $d(x, z) \leq d(x, y) + d(y, z)$ (the "triangle" inequality).

As mentioned before we will define completeness in terms of collections of *closed balls*. These closed balls consist of a (centre) point and a (positive) radius. We consider two properties:

Consistency We say that a collection of balls is consistent if for any balls (x, r) (y, s) we have that $d(x, y) < r + s$. Note here we *don't* assume that some point exists in the intersection.

Arbitrarily Small Radii A collection of balls has this property if there exist balls with arbitrarily small radi.

A metric space is then said to be *complete* if every consistent set of balls with arbitrarily small radii has a limit, i.e. the intersection of the balls is non-empty. (If it exists, it must be unique.) We can convert a collection of closed balls into a Cauchy sequence by taking x_i to be a ball of radii less than 2^{-i}, and vice versa.

Theorem If M is any metric space, then there is a complete metric space $Comp(M)$ with an injection $i: M \mapsto Comp(M)$ such that the metric in M is preserved ($d(x, y) = d(i(x), i(y))$) and furthermore the image of M under i is dense in $Comp(M)$.

Here we must define dense not in the usual topological sense but in terms of a metric. We say a subset X of a metric space M is *dense* if for any point y of M and $\epsilon > 0$, there is a point x in X with $d(x, y) < \epsilon$.

We note that the completion can be shown to be unique up to isomorphism by metric preserving maps.

Proof Take a metric space M, the points in the completion are the consistent families of balls with arbitrarily small radius. We define the distance between two such families F and G as the limit of the intervals

$$[d(x, y) - (r + s), d(x, y) + (r + s)]$$

where (x, r) ranges over the balls in F and (y, s) over the balls in G. The only tricky thing is showing that any pair of intervals is consistent; here a pair of intervals $[a, b]$ and $[c, d]$ are consistent if $\max(a, c) \geq \min(b, d)$ so the two intervals overlap. In this case this is equivalent to showing that for any (x_1, r_1),(x_2, r_2) in F and (y_1, s_1),(y_2, s_2) in G,

$$|d(x_1, y_1) - d(x_2, y_2)| \leq (r_1 + s_1) + (r_2 + s_2)$$

but since F and G are consistent we know that $d(x_1, x_2) \leq r_1 + r_2$ and $d(y_1, y_2) \leq s_1 + s_2$ then the above inequality can be derived using the triangle inequality. It is trivial to see that this collection of intervals has arbitrarily small lengths. Therefore the limit of these intervals exists and is unique. Then we further have to show that this distance function satisfies the axioms; this follows straightforwardly from the fact that the original metric d satisfies the axioms.

We also note that given any x in M, the collection of balls $\{(x, 0)\}$ is consistent and has balls of arbitrarily small radius; this is the injection of x into the completion. It is trivial that the injection preserves the metric. Then it is easy to show that the image of M is dense in the completion. For any point in the completion, say F, which is a collection of consistent closed balls with arbitrarily small radius and for any $\epsilon > 0$ we need to find a point in M whose image is less than ϵ away from F. We just take x where (x, r) is in F with $r \leq \epsilon$. Then the distance between F and $\{(x, 0)\}$ lies in the interval $[d(x, x) - (r + 0), d(x, x) + (r + 0)]$, i.e. it is less than r as required.

Note the similarities between this proof and the previous one. In both cases we use the basic two notions of consistency and arbitrarily smallness. But because we use a different basic type (intervals in one case and balls in the other) the algebra is very different. Because of this it is not easy to re-use parts of the two proofs.

3 Implementing the Mathematics

In this section we will describe how these mathematical ideas above were implemented, and how far the whole experiment has got. We will note how some features of LEGO were particularly useful and also some of the problems with using LEGO.

3.1 Basic Methods

We implemented this mathematics using many claimed lemmas, rather than starting from the bottom and working up.

In the metric space example I assumed the existence of a set "real", and the various operators and relations usually considered on it (addition, multiplication, less than etc.). But I didn't attempt to axiomatise the operations (i.e. say that addition forms an Abelian group with zero as identity) but I merely claimed all the lemmas as I needed them, and in the form I needed them. I did a similar thing with the rationals in the case of defining the reals. In both cases I assumed only lemmas which could be proved fairly simply by algebraic manipulation of the basic axioms.

Here are some lemmas which are typical of the sort of thing I assumed. The first three came from the reals, the last is for the rationals.

`half` For all $x > 0$ there exists $y > 0$ s.t. $x \geq y + y$.

`permute_4` For all a, b and c, $(a + (b + c)) = (c + (a + b))$. There are other ones similar to this!

`lesseq_triv` If $a \geq 0$ then $(x + a) \geq (x - a)$.

`dense_R` For all a and b, if for all $x > 0$, $a \leq b + x$ then $a \leq b$.

There were several pragmatic reasons for doing this. Firstly when this work as started, there was no satisfactory representation of the natural numbers as a datatype. The existing methods made it very tedious to prove elementary facts such as the distributivity of multiplication over addition. Since then the system has been extended to make such proofs easier, although a completely satisfactory account of general recursive datatypes is still a research topic. Secondly it was felt that we could make progress by assuming lemmas and perhaps we would not in fact need all that many different lemmas. This did prove to be the case in the end: we used only about 30 lemmas of mostly algebraic facts about the real numbers in the metric space example. Thirdly we felt that the lemmas we generated would perhaps be better solved in the context of a tactics or automated theorem prover, rather than some considerable human effort.

We have considered the problem of proving the lemmas we used about the reals. One possibility is to express them restricted to the rationals. Then there clearly is some general theorem saying that an equality (or inequality, but not strict inequality) of two expressions involving the obvious set of operators which holds in the rational also holds in the reals when all the operators are extended. We cannot formulate and prove this notion within our system as it stands; or rather we need some reflection principle to apply it to get the instances we need. The other possibility is to take some standard minimal but complete axiomatisation and attempt to prove all the lemmas we used from that. This would probably be a rather tedious exercise, which the author would prefer not to undertake.

3.2 Representing sets and other mathematics in ECC

We used a common file for both examples which contained a definition of the mathematical entities like sets, subsets and functions. The basic idea is that sets are just types with extra structure. We choose to represent a set as a type with a total equivalence relation over it which is the equality for elements of that set. This is fairly similar to the notion of set used by Maharaj [10]. We do not need to use PERs in order to restrict our sets (i.e. to consider the set of positive rationals) as Coquand does in pure CC since we can use sigma types. Thus we distinguish between the two notions of firstly *restricting* to a smaller set and secondly *quotienting* by some equivalence relation thus *identifying* some of the elements.

The sigma type device lets us consider positive rationals as the type consisting of pairs, firstly a rational, secondly the proof that the rational is positive. We use this idea again with subsets. We consider a subset of a set to be a predicate on the underlying type of a set. If we wish to consider this subset as a set in its own right, we form the sigma type whose first part is an element of parent type, the second a proof that this element satisfies the predicate. Thus we can apply operations like powerset and product to *sets* and operations like intersection and union to *subsets* and moreover the subsets must be subsets of the same set. This seems to reflect the use of set theory in many areas of mathematics, although it is contrary to most axiomatisations of set theory which allow e.g. the intersection of unrelated sets.

Whether this is the best method for representing sets in some GTS framework is still a research problem. Pragmatically the representation I have chosen has been sufficient for the examples studied.

We also use sigma types in packing together the types, operations and axioms which make up a mathematical structure. For example, a set consists of a single type, a relation over that type (equality) and the three axioms that say this relation is an equivalence. In LEGO we code this as:

```
[A:Type];
[eq:A -> A -> Prop];
eq_refl == {x:A}eq x x;
eq_sym == {x,y:A}(eq x y) -> (eq y x);
eq_trans == {x,y,z:A}(eq x y) -> (eq y z) -> (eq x z);
Discharge eq;
eq_rel == <eq:A->A->Prop>and3 (eq_refl eq) (eq_sym eq) (eq_trans eq);
Discharge A;
set == <A:Type>eq_rel A;
```

The first two lines are declarations which add to the context and are discharged later in the example. The braces `{...}` represent a π-binding and `<...>` a sigma. Thus we can declare e.g. `A` to be a set with `A:set`. Then the projections of the sigma will give us the carrier type, the relations and the axioms.

In fact we have a very nice general trick here which depends on the LEGO implicit typing system. We can refer to the "generic" equality on sets by the definition:

```
[A|set];
eq == A.2.1;
Discharge A;
```

So now `eq x y` is typable in any context where `x` and `y` are elements of the carrier of some set and lego will find the appropriate relation by matching. In effect we can talk about the equality of two elements of a set without having to specify the equality we are talking about since it is implied by the set. This is very much the way mathematics is written, where the $=$ symbol is used freely and the reader infers which equality is appropriate from the context. This trick can be carried further in that we can also name and use the generic equality axioms. The implicit types act a bit like polymorphism, although the matching used to instantiate them is sometimes weaker than SML polymorphism.

3.3 Using Lemmas—An Example

We will look at a specific problem with doing this sort of proof, describe the solution and say what lessons we can learn from this.

The example we consider is in defining multiplication. We multiply two intervals $[a, b]$ and $[x, y]$ to get the result interval $[\min(ax, ay, bx, by), \max(ax, ay, bx, by)]$. The reason we have to put max and min in is two-fold. Firstly intervals are *ordered* pairs with the second element greater than the first, secondly in the case when some of the numbers are negative, we cannot assume that by is the greatest product. We use the expected axioms about $\min(x, y)$ and $\max(x, y)$: essentially that they are the greatest lower bound and least upper bound of the set $\{x, y\}$.

One of the lemmas we have to prove is that given two pairs of overlapping intervals the products overlap. Say we have $[a, b]$, $[c, d]$ and $[x, y]$,$[w, z]$ with $[a, b]$ overlapping with $[c, d]$ and $[x, y]$ overlapping with $[w, z]$ then we need to show that the product of $[a, b]$ with $[x, y]$ will overlap with the product of $[c, d]$ with $[w, z]$. Here we say two intervals $[a, b]$ and $[c, d]$ overlap if $\max(a, c) \leq \min(b, d)$. So what we have to prove is the equation

$$\max(\min(ax, ay, bx, by), \min(cz, cw, dz, dw)) \leq \min(\max(ax, ay, bx, by), \max(cz, cw, dz, dw))$$

from $\max(a, c) \leq \min(b, d)$ and $\max(x, z) \leq \min(y, w)$. Note we have already simplified this from the lego syntax since officially min and max have arity 2. We can simplify this somewhat into the two cases

$$\begin{aligned} \min(ax, ay, bx, by) &\leq \max(cz, cw, dz, dw) \\ \min(cz, cw, dz, dw) &\leq \max(ax, ay, bx, by) \end{aligned}$$

but from here on it is not easy to see how to proceed. We clearly want to avoid considering a complicated case analysis of whether all the 8 variables are positive and negative and which ones are larger.

So instead we reflect a little: first we consider working from the "other" end of the problem. We will clearly need some sort of lemma about how multiplication interacts with $\leq$. We want to avoid lemmas with conditions like the obvious lemma $0 \leq x$ implies $a \leq b$ implies $ax \leq bx$. So a good choice for a reasonably simple lemma is the following

$$a \leq x \leq b \Rightarrow \forall z \ \min(az, bz) \leq xz \leq \max(az, bz).$$

In fact we can prove this from the lemma above by considering only the two cases $z \leq 0$ and $0 \leq z$.

To make the theorem go really smoothly, we now consider a little fact about overlap. Another way of looking at overlapping intervals is to say that they have a common point. If we can prove this is equivalent to the usual definition, then we only have to prove that if a point k is in the interval $[a, b]$ and l is in $[x, y]$ then kl is in the product of $[a, b]$ and $[x, y]$. This can be done with two applications of the lemma above (and multiplication symmetry).

Now at first sight, showing that overlapping intervals have a point in common seems a bit dubious. What about all those proofs where we showed that two intervals overlapped by using the lemma "Dense R" in section 3.1, page 214? But on reflection we can see immediately that from a proof of overlap of $[a, b]$ and $[c, d]$, we already have *two* common points, namely $\max(a, c)$ and $\min(b, d)$. It is easy to that these are common points. This general trick is quite useful in constructive mathematics, since we can construct the functions min and max on the reals so we can refer to "the greater of x and y" although we can't say either x is the greater or y is the greater. We also see that the apparently very powerful lemma "dense R" can just be shifted down, we can give a definition of the overlapping point, the proof that it is in each of the intervals requires use of "dense R" again.

So we can get the proof we need by using the above lemma and the transitivity of less-than or equals on a mid-point which might be (for instance) $\max(a, c) \max(x, z)$.

So what can we learn from this episode: mainly that finding the right lemma can save you a significant amount of algebraic manipulation. We note the crucial thing about this lemma was that it reflected the problem we were working on, in that it incorporated the idea of using min and max to avoid worrying about sign changes. Applying the methods above to the next part of the definition of multiplication is not quite so successful. We have to bound the length of the product interval, i.e. do the algebra

$$(ac - bd) = a(c - d) + (a - b)d$$

but for every possible length of interval, i.e. we have 16 different cases, 4 of which are of the form $(ac - ac)$, and a further 4 are of the form $(ac - bc)$ and the rest are as above. There does not seem to be any way to simplify this substantially. We can't get a lemma which will solve all of these because we have to sometimes use symmetry of $+$ and we have to know that both ends of a given interval are bounded by the given upper bound.

3.4 Using Lemmas—problems

There are times when the way in which a lemma is expressed can make life a lot harder than it ought to be. For instance it is almost never a good idea to put the two parts of a if-and-only-if statement together. This is because we will almost invariably want to refer to each part separately. However in lego the way the implicit syntax works, we can't easily refer to the first and second parts, but we need to know how many implicit variables lie outside the iff part. What happens it that $\leftrightarrow$ is defined in terms of `and`, for which there are the two "polymorphic" destructors `fst` and `snd`. If we have an axiom such as

```
null == {x,y|A.1}(iff (eq x y) (eq (d.1(x,y)) ze));
```

then we cannot refer to the implication one way as `fst null` since we have quantified implicitly by `x` and `y` as well. We need to say `fst (null|?|?)`, so although we do not need to give the arguments we still need to know how many of them there are.

A further example is the axiom which stated that if a given interval was in a convergent collection of intervals, then the limit of the convergent set was inside the interval (Completeness property of the reals). For this axiom it was necessary to explicitly give the interval rather then the more natural requirement to show that there was an interval with the expected end-point. It was also difficult to give the interval by using a "claim" which could later be resolved since "claim interval" was too general to match the goal. It would have been possible to claim the various subgoals of interval; one of which would have been resolved by matching against the completeness axioms, but this was rather clumsy. Another possibility would have been to re-write the axiom as the less natural:

```
{x:Re}{F:ok_ints}{I:interval}(F I) -> (eq I.1 x)
-> (lesseq x (lim F));
```

and

```
{x:Re}{F:ok_ints}{I:interval}(F I) -> (eq I.2.1 x)
-> (lesseq (lim F) x);
```

instead. Then these axioms could have been applied to produce three new subgoals: the interval, that the interval was in F and the condition on x. We could have used `next` and `intros#` and a refine by equal reflexivity to resolve all references to the term which will be substituted for by "x". We want to avoid writing axioms in this unnatural way, since it is then much harder to check that they say the right thing. Perhaps the best answer is to prove these two facts as lemmas from the original axioms.

3.5 The Results

So far we have completed the metric space example. This example took approximately 1,300 lines of lego code involving approximately 800 refinement operations. It takes about 350 secs to run on a Sun 4/390. The completion of the reals is not yet finished. We have defined the set, and the operation of addition and the order relations but we haven't done multiplication or the completeness proof yet. We hope to finish this example in a fairly short time. What we have done has about 1,000 lines and 450 refinement operations.

4 Conclusion

So we have seen that it is possible to verify quite complicated mathematics within a fairly simple system. We have seen not only that calculus of constructions is expressible enough to represent this part of mathematics, but also that we can do so in a fairly natural way. The combination of using sigma types to package up structures and implicit typing to give us generic names for the parts of these structures is particularly powerful.

We hope soon to complete the work described here: in particular to finish the construction of the reals from the rationals. We would need to do a lot more work to get a complete proof i.e. with all the lemmas proved. It would be nice to do this and to develop the rationals from a natural number datatype.

As we have discussed, often apparently simple informal reasoning (particularly algebraic reasoning with equality etc.) can require extremely long and tedious proofs. I believe there probably isn't any single answer to this sort of problem. Perhaps some more specific tactics would help to solve a collection of similar goals. The idea of symmetry seems to be very important: we often find proofs come in repeated blocks where we perhaps interchange two variable names and use different hypotheses. In lego we usually prove these goals by "cutting and pasting" from a text editor. When such a block requires two or three hypothesis it is often too much trouble to express it as a formal lemma, because the context of the string of deductions is longer than the actual deductions. Perhaps there should be some way to extract the lemma by taking a solved goal and abstracting on variables and hypotheses which are used in its proof. Probably it would not be possible to make this an wholly automatic process. Another possibility is to consider making lemmas inside part of the proof context, this would only work up to a point. If we want to interchange the role of "x" and "y" then obviously the lemma must generalise both, not take them from the context.

I have noticed that the "refine" tactic in LEGO (which is used to make most proof steps, the other main tactic is intros which does pi or sigma introductions) encourages the style of refinement proofs where we always work on the goal, rather than building something up from the hypotheses. We can use "claim" to change this to some degree. I feel that this style of proof is better suited to constructive rather than classical reasoning.

References

[1] Lennart Augustsson, Thierry Coquand, and Bengt Nordstrom. A short description of another logical framework. In *Informal Proceedings of the First Annual Workshop on Logical Frameworks*, 1990.

[2] E. Bishop. *Foundations of Constructive Analysis.* McGraw-Hill, 1967.

[3] Robert Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[4] Thierry Coquand and Gerard Huet. *Constructions: a Higher Order Proof System for Mevchanising Mathematics*, volume 203 of *LNCS*. Springer-Verlag, 1985. EUROCAL85, Linz.

[5] N. G. DeBruijn. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms*, chapter A Survey of the Project Automath. Academic Press, 1980.

[6] Gerard Huet, Thierry Coquand, Gilles Dowek, and Christine Paulin-Mohring. The calculus of constructions. documentation and user's guide. Technical Report 110, INRIA, 1989.

[7] Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, Department of Computer Science, University of Edinburgh, 1990.

[8] Zhaohui Luo, Randy Pollack, and Paul Taylor. How to use LEGO (a preliminary user's manual). Technical report, University of Edinburgh, 1989.

[9] S. MacLane and G. Birkoff. *Algebra.* Macmillan, 1979.

[10] Savitri Maharaj. Implementing Z in LEGO. Master's thesis, Department of Computer Science, University of Edinburgh, 1990.

[11] Gabriel Stolzenberg. Notes on constructive analysis. In preparation.

A Appendix: List of Lemmas

A.1 Metric Space lemmas

We assume the following types, functions and relations.

```
Real : set
zero : Real
one : Real
plus : Real x Real -> Real
subtract : Real x Real -> Real
mult : Real x Real -> Real
min : Real x Real -> Real
max : Real x Real -> Real
lesseq : Real x Real -> Prop
lessst : Real x Real -> Prop
```

Axioms involving these datatypes:

`plus_sym`	$(x + y) = (y + x)$
`lesseq_if`	$\forall\, z > 0,\ x \leq (y + z + z)) \Rightarrow x \leq z$
`sort_plus`	$(x + z) + (z + y) = ((x + y) + (z + z))$
`lesseq_trans`	$(x \leq y) \Rightarrow (y \leq z) \Rightarrow (x \leq z)$
`lesseq_plus_r`	$x \leq y \Rightarrow (x + z) \leq (y + z)$
`lesseq_plus_l`	$x \leq y \Rightarrow (z + x) \leq (z + y)$
`lesseq_min`	$(x \leq \min(y, z)) \leftrightarrow (x \leq y)\&(x \leq z)$
`lesseq_max`	$(\max(x, y) \leq z) \leftrightarrow (x \leq z)\&(y \leq z)$
`lesseq_subl`	$(x - y) \leq z \leftrightarrow x \leq (z + y)$
`lesseq_subr`	$x \leq (y - z) \leftrightarrow (x + z) \leq y$
`lesseq_triv`	$0 \leq y \Rightarrow (x - y) \leq (x + y)$
`lesseq_incl_eq`	$x = y \Rightarrow x \leq y$
`lesseq_imp_eq`	$x \leq y \Rightarrow y \leq x \Rightarrow x = y$
`plus_perm`	$(b + d) + (a + (c + e)) = (a + ((b + c)) + (d + e))$
`eq_sub`	$x - y = z \leftrightarrow x = z + y$
`plus_ze`	$x = x + 0$
`lessst_imp`	$x < y \Rightarrow x \leq y$
`sub_gives_pos`	$x \leq y \Rightarrow 0 \leq y - x$
`lesseq_plus_all`	$(x \leq a) \Rightarrow (y \leq b) \Rightarrow (x + y) \leq a + b$
`half`	$\forall\, x > 0,\ \exists\, y > 0,\ (y + y) < x$
`sum_pos`	$x \geq 0 \Rightarrow y \geq 0 \Rightarrow x + y \geq 0$
`pos_plus_lesseq`	$x \leq y \Rightarrow 0 \leq z \Rightarrow x \leq y + z$
`eq_subl`	$(x - y) = z \leftrightarrow x = z + y$
`eq_subr`	$x = (y - z) \leftrightarrow (x + z) = y$
`plus_sub_perm`	$(a + b) + (c - d) = a + ((b + c) - d)$
`permute_1`	$(a - c) + b = a + (b - c)$
`by_six`	$\forall\, a > 0,\ \exists\, x > 0,\ (x + x + x + x + x + x) \leq a$
`perm3`	$(v + (x + x) - (r + s) + (w + (x + x) - (s + t) + (x + x) = (x - (r + s)) + (w - (s + t)) + (x + x + x + x + x + x)$
`permute_4`	$(a + (b + c)) = (c + (a + b))$

We define the type `Interval` and the properties of consistency and arbitrarily smallness on sets of intervals and call the resulting type `ok_ints`. We define the function

```
lim: ok_ints -> Real
```

and then can give some axioms.

`complete_re`	For an interval I and an element of `ok_ints` F, then the left-hand endpoint of I (left(I)) and the right-hand endpoint (right(I)) satisfy $\mathrm{left}(I) \leq \lim(F) \leq \mathrm{right}(I)$
`eq_if`	if F and G are elements of `ok_ints` and are consistent with each other then $\lim(F) = \lim(G)$

We define the sum of two collection of intervals (as in the definition of multiplication).

`sum_is_ok`	If F and G are `ok_ints` then so is $F + G$.
`plus_com_with_lim`	$\lim(F + G) = \lim(F) + \lim(G)$

A.2 Lemmas for defining the reals from the rationals

Declare the following types, functions and relations.

```
Rat : set
zero : Rat
one : Rat
plus : Rat x Rat -> Rat
subtract : Rat x Rat -> Rat
mult : Rat x Rat -> Rat
divide : Rat x (positive Rat) -> Rat
min : Rat x Rat -> Rat
max : Rat x Rat -> Rat
lesseq : Rat x Rat -> Prop
lessst : Rat x Rat -> Prop
```

`plus_sym`	$x + y = y + x$
`min_sym`	$\min(x, y) = \min(y, x)$
`max_sym`	$\max(x, y) = \max(y, x)$
`lesseq_triv`	$\min(x, y) \leq \max(x, y)$
`lesseq_max`	$x \leq z \Rightarrow y \leq z \Rightarrow \max(x, y) \leq z$
`lesseq_min`	$x \leq y \Rightarrow x \leq z \Rightarrow x \leq \min(y, z)$
`lesseq_plus`	$x \leq y \Rightarrow (x + z) \leq (y + z)$
`lesseq_sub`	$(x - y) \leq z \leftrightarrow x \leq (z + y)$
`lesseq_trans`	$x \leq y \Rightarrow (y \leq z) \Rightarrow (x \leq z)$
`lesseq_max_rev`	$\max(x, y) \leq z \Rightarrow x \leq z \& y \leq z$
`lesseq_min_rev`	$x \leq \min(y, z) \Rightarrow x \leq y \& x \leq z$
`lesseq_eq`	$x \leq x$
`subtract_equals`	$x - x = 0$
`lesst_imp_lesseq`	$x < y \Rightarrow x \leq y$
`lesseq_imp_min_max`	$x \leq y \Rightarrow \min(x, y) = x \& \max(x, y) = y$
`min_imp_lessl`	$\min(x, y) \leq x$
`min_imp_lessr`	$\min(x, y) \leq y$
`max_imp_lessl`	$x \leq \max(x, y)$
`max_imp_lessr`	$y \leq \max(x, y)$
`divide_eq`	$y/x \times x = y$
`dense_R`	$\forall z > 0,\ x \leq y + z \Rightarrow x \leq y$
`lesseq_plus_2`	$(a \leq b) \Rightarrow (c \leq d) \Rightarrow (a + c) \leq (b + d)$

`half: Rat -> Rat`

`half_eq`	$x = \mathrm{half}(x) + \mathrm{half}(x)$
`half_pos`	$0 \leq x \Rightarrow o \leq \mathrm{half}(x)$
`perm_11`	$(a+b) - (c+d) = (a-c) + (b-d)$
`min_max2`	$a \leq b \Rightarrow c \leq d \Rightarrow \min(a,c) \leq \max(b,d)$
`mult_is_mono`	$a \leq x \leq b \Rightarrow \min(ay, by) \leq xy \leq \max(ay, by)$
`mult_is_mono_op`	$a \leq x \leq b \Rightarrow \min(ya, yb) \leq yx \leq \max(ya, yb)$
`divide_pos`	$0 < x \Rightarrow 0 < x/y$

We define the function mod (modulus) and assume the following.

`mod_plus`	$\lvert x+y\rvert \leq \lvert x\rvert + \lvert y\rvert$
`mod_mult`	$\lvert xy\rvert \leq \lvert x\rvert\lvert y\rvert$
`mod_sym`	$\lvert x-y\rvert = \lvert y-x\rvert$
`min_max_subtract`	$\max(a,b) - \min(x,y) = \max(\max((a-x),(a-y)), \max((b-x),(b-y)))$
`min_max_sub_simp`	$\max(a,b) - \min(a,b) = \max((a-b),(b-a))$
`algebra_1`	$ac - bd = (a-b)c + b(c-d)$
`mod_less`	$x \leq \lvert x\rvert$
`lesseq_plus_2`	$x \leq y \Rightarrow a \leq b \Rightarrow (x+a) \leq (y+b)$
`pos_mult_lesseq`	$0 \leq a \Rightarrow a \leq c \Rightarrow 0 \leq b \Rightarrow b \leq d \Rightarrow ab \leq cd$
`mod_pos`	$0 \leq \lvert x\rvert$

Negation versus Implication

Yves LAFONT
Laboratoire d'Informatique
Ecole Normale Supérieure *
(URA 1327 du CNRS)

22 June 1991

Abstract

This sketchy note comes from my attempt to understand Girard's recent work on making classical logic constructive [Girard91]. I was interested in the system itself, which can be seen as a programming language, rather than in extracting the computational content of classical proofs.

Keywords: *classical logic, continuation passing style, double-negation translation, intuitionistic logic, functional programming, sequent calculus.*

1 Calculus

We consider a propositional calculus with sequents of the form $\Gamma \vdash \Pi$ where Γ is an arbitrary sequence of formulae and Π consists of one or zero formula. Formulae are of the form P (atom), $A \wedge B$ (conjunction) or $\sim A$ (negation). Later we shall introduce a disjunction, but we insist on the absence of implication. The rules are basically the same as for intuitionistic logic (Gentzen's LJ) except that the right side of a sequent may be empty.

- structure:

$$\frac{\Gamma, A, B, \Delta \vdash \Pi}{\Gamma, B, A, \Delta \vdash \Pi} \qquad \frac{\Gamma, A, A \vdash \Pi}{\Gamma, A \vdash \Pi} \qquad \frac{\Gamma \vdash \Pi}{\Gamma, A \vdash \Pi}$$

- identity and cut:

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash \Pi}{\Gamma, \Delta \vdash \Pi}$$

- conjunction:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma, A, B \vdash \Pi}{\Gamma, A \wedge B \vdash \Pi}$$

*Address: 45 rue d'Ulm, 75230 Paris Cedex 05, France. Email: lafont@dmi.ens.fr

- negation:

$$\frac{\Gamma, A \vdash}{\Gamma \vdash \sim A} \qquad \frac{\Gamma \vdash A}{\Gamma, \sim A \vdash}$$

We can already make the following comments:

- This calculus satisfies the *cut elimination property*: any proof using cuts can be transformed into a cut-free proof of the same sequent.
- The negation here is *nonclassical* in the sense that $\sim\sim P \vdash P$ is not provable (obvious by the cut elimination property), but it is even *minimal* in the sense that $P, \sim P \vdash Q$ is not provable (again by cut elimination). However $P, \sim P \vdash$ is provable:

$$\frac{\overline{P \vdash P}}{P, \sim P \vdash}$$

This calculus can be seen as a fragment of intuitionistic logic. Fix indeed an arbitrary formula Φ and consider $\Gamma \vdash$ (resp. $\sim A$) as a notation for $\Gamma \vdash \Phi$ (resp. $A \Rightarrow \Phi$). The calculus above is obviously sound for this interpretation. The rules of negation are interpreted as follows:

$$\frac{\Gamma, A \vdash \Phi}{\Gamma \vdash A \Rightarrow \Phi} \qquad \frac{\Gamma \vdash A \qquad \overline{\Phi \vdash \Phi}}{\Gamma, A \Rightarrow \Phi \vdash \Phi}$$

Completeness, *i.e.* the fact that a sequent $\Gamma \vdash \Pi$ is provable in the above calculus whenever its interpretation (with an atomic Φ) is provable in LJ, is not completely obvious. We leave this point as an exercise.

In practice, we shall rather use a calculus with *implicit structural rules*, closer to natural deduction:

- identity and cut:

$$\frac{}{\Gamma, A, \Delta \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Gamma, A \vdash \Pi}{\Gamma \vdash \Pi}$$

- conjunction:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma, A, B, \Delta \vdash \Pi}{\Gamma, A \wedge B, \Delta \vdash \Pi}$$

- negation:

$$\frac{\Gamma, A \vdash}{\Gamma \vdash \sim A} \qquad \frac{\Gamma, \sim A, \Delta \vdash A}{\Gamma, \sim A, \Delta \vdash}$$

This calculus is clearly equivalent to the previous one, and it satisfies the cut elimination property as well. We introduce a notation for proofs using the following syntax:

$$
\begin{array}{lcl}
\textit{expression} & : & \textit{term} \\
 & | & \textit{command} \\
\\
\textit{term} & : & \textit{variable} \\
 & | & \textbf{let } \textit{pattern} = \textit{term} \textbf{ in } \textit{term} \\
 & | & (\textit{term}, \textit{term}) \\
 & | & \textit{pattern} \rightarrow \textit{command} \\
\\
\textit{pattern} & : & \textit{variable} \\
 & | & (\textit{pattern}, \textit{pattern}) \\
\\
\textit{command} & : & \textbf{let } \textit{pattern} = \textit{term} \textbf{ in } \textit{command} \\
 & | & \textit{variable}\ \textit{term}
\end{array}
$$

A *term* denotes a proof of a sequent $\Gamma \vdash A$, whereas a *command* denotes a proof of $\Gamma \vdash$. Now, the calculus becomes an inference system for typing judgements of the form $\Gamma \vdash e : \Pi$ where Γ is a sequence of typed patterns $p : A$ and Π consists of one formula (in which case e is a term) or zero formula (in which case e is a command).

- variable and local binding:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma, p : A \vdash e : \Pi}{\Gamma \vdash \textbf{let } p = a \textbf{ in } e : \Pi}$$

- pair and split:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \wedge B} \qquad \frac{\Gamma, p : A, q : B, \Delta \vdash e : \Pi}{\Gamma, (p, q) : A \wedge B, \Delta \vdash e : \Pi}$$

- abstraction and application:

$$\frac{\Gamma, p : A \vdash c :}{\Gamma \vdash p \rightarrow c : \sim A} \qquad \frac{\Gamma, x : \sim A, \Delta \vdash a : A}{\Gamma, x : \sim A, \Delta \vdash x\, a :}$$

Notice the following differences with λ-calculus:

- The right member of an abstraction $p \rightarrow c$ is a command, in particular it cannot be a variable or an abstraction.
- The left member of an application $x\, a$ is a variable, in particular it cannot be an application or an abstraction.
- Local binding and explicit pairs are needed. In fact, a term with no local binding is in normal form.

The analogue of β-reduction is not obvious, but we have a clear idea of what is the evaluation mechanism (or an abstract machine) for this calculus. Since a command does not return a value, no stack is needed in the implementation.

2 Translation

The above calculus seems quite bare, but it is in fact as powerful as simply typed λ-calculus. For that purpose, we exhibit what is known as a *continuation passing style* transformation [Fisher72]. More precisely, we translate formulae builded from atoms $P, Q, \ldots$ using only implication into formulae builded from atoms $P^*, Q^*, \ldots$ using only conjunction and negation. Intuitively P^* corresponds to the *negation* of P.

Star is extended inductively to composed formulae as follows:

$$(A \Rightarrow B)^* = \sim A^* \wedge B^*$$

For intuitionistic logic, we use the following calculus:

- structure:

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C} \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}$$

- identity and cut:

$$\frac{}{A \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}$$

- implication:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C}$$

Proposition 1 *If $A_1, \ldots, A_n, B$ are formulae builded from atoms $P, Q, \ldots$ using only implication and $A_1, \ldots, A_n \vdash B$ is provable intuitionistically then $\sim A_1{}^*, \ldots, \sim A_n{}^*, B^* \vdash$ is provable in our calculus.*

Proof: By induction on proofs, using the fact that the following rules are derivable:

$$\frac{}{\sim A, A \vdash} \qquad \frac{\Gamma, A \vdash \qquad \Delta, \sim A \vdash}{\Gamma, \Delta \vdash} \qquad \frac{\Gamma, \sim A, B \vdash}{\Gamma, \sim A \wedge B \vdash} \qquad \frac{\Gamma, A \vdash \qquad \Delta, \sim B \vdash}{\Gamma, \Delta, \sim(\sim A \wedge B) \vdash}$$

This is straightforward, except for the last case, where a cut is needed (we skip exchange):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, A \vdash}{\Gamma \vdash \sim A} \qquad \dfrac{}{B \vdash B}}{\Gamma, B \vdash \sim A \wedge B}}{\Gamma, \sim(\sim A \wedge B), B \vdash}}{\Gamma, \sim(\sim A \wedge B) \vdash \sim B} \qquad \Delta, \sim B \vdash}{\Gamma, \Delta, \sim(\sim A \wedge B) \vdash}$$

Q.e.d.

The converse of this result does not hold: in fact we shall see that $\sim A_1{}^*, \ldots, \sim A_n{}^*, B^* \vdash$ is provable in our calculus if and only if $A_1, \ldots, A_n \vdash B$ is provable *classically.*

It is also interesting to express this translation at the level of terms. For each λ-term M of type B with free variables $x_1, \ldots, x_n$ of types $A_1, \ldots, A_n$, we define inductively a pattern M^* and a command M_* such that

$$x_1 : \sim A_1^*, \ldots, x_n : \sim A_n^*, M^* : B^* \vdash M_* :$$

is derivable in our calculus.

variable:	abstraction:	application:
$x^* \;=\; y$ (fresh variable)	$(\lambda x.M)^* \;=\; (x, M^*)$	$(M\,N)^* \;=\; y$ (fresh variable)
$x_* \;=\; x\,y$	$(\lambda x.M)_* \;=\; M_*$	$(M\,N)_* \;=\; \textbf{let } M^* = (N^* \to N_*, y) \textbf{ in } M_*$

Just looking at the case of application, we guess that this translation provides a call-by-name implementation of λ-calculus.

Now we present Girard's translation of classical logic. First we must add a disjunction to our calculus, with the following rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma, A \vdash \Pi \qquad \Gamma, B \vdash \Pi}{\Gamma, A \vee B \vdash \Pi}$$

Classical formulae are of the form P (positive atom), $\neg A$ (negation), $A \wedge B$ (conjunction), $A \vee B$ (disjunction) or $A \Rightarrow B$ (implication). For each classical formula A, we define a *sign* $\varepsilon(A)$ and a formula $|A|$ in our calculus, according to the following tables:

$\varepsilon(P)$	$\lvert P\rvert$
$+$	P

$\varepsilon(A)$	$\varepsilon(\neg A)$	$\lvert\neg A\rvert$
$+$	$-$	$\lvert A\rvert$
$-$	$+$	$\lvert A\rvert$

$\varepsilon(A)$	$\varepsilon(B)$	$\varepsilon(A \wedge B)$	$\lvert A \wedge B\rvert$	$\varepsilon(A \vee B)$	$\lvert A \vee B\rvert$	$\varepsilon(A \Rightarrow B)$	$\lvert A \Rightarrow B\rvert$
$+$	$+$	$+$	$\lvert A\rvert \wedge \lvert B\rvert$	$+$	$\lvert A\rvert \vee \lvert B\rvert$	$-$	$\lvert A\rvert \wedge \sim\lvert B\rvert$
$+$	$-$	$+$	$\lvert A\rvert \wedge \sim\lvert B\rvert$	$-$	$\sim\lvert A\rvert \wedge \lvert B\rvert$	$-$	$\lvert A\rvert \wedge \lvert B\rvert$
$-$	$+$	$+$	$\sim\lvert A\rvert \wedge \lvert B\rvert$	$-$	$\lvert A\rvert \wedge \sim\lvert B\rvert$	$+$	$\lvert A\rvert \vee \lvert B\rvert$
$-$	$-$	$-$	$\lvert A\rvert \vee \lvert B\rvert$	$-$	$\lvert A\rvert \wedge \lvert B\rvert$	$-$	$\sim\lvert A\rvert \wedge \lvert B\rvert$

The sign has got nothing to do with a truth value (a positive atom can be true or false). Now we define the translation $\langle A\rangle$ as follows:

$$\begin{aligned} \langle A\rangle &= |A| \quad \text{if } \varepsilon(A) = + \\ &= \sim|A| \text{ if } \varepsilon(A) = - \end{aligned}$$

Proposition 2 *If $A_1, \ldots, A_n, B_1, \ldots, B_m$ are classical formulae then $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ is provable classically if and only if $\langle A_1\rangle, \ldots, \langle A_n\rangle, \langle\neg B_1\rangle, \ldots, \langle\neg B_m\rangle \vdash$ is provable in our calculus.*

Unlike usual double-negation translations, Girard's one is compatible with substitution (by positive formulae only), and it preserves canonical isomorphisms like the involutivity of negation or the associativity of disjunction. It is also more economical in the sense that negations are only introduced as a last resort. It extends to first-order logic (see [Girard91, Gallier91]).

Our translation of the implicational fragment is just a special case, if we consider formulae builded from negative atoms $\neg P$ and using only implication.

3 Programming

We have seen that λ-calculus can be embedded into a calculus with negation types instead of implications. It is well known that this sort of translation extends to nonfunctional features like *side effects*, *exceptions* and *continuations*. This is used, for example, in the implementation of *Standard ML of New Jersey*.

Here we propose another approach: why not programming directly in the restricted language? This may seem cumbersome at the beginning, but we gain referential transparency, which could be useful at least for a pedagogical purpose. Let us see little examples to get an idea of what is happening. In each case, we compare a program written in CAML with its continuation passing style analogue. For the syntax we refer to [CAML].

To start with, consider the trivial example of the identity function. We enter the program and the system infers the (most general) type:

```
#let id = function x -> x;;
Value id = <fun> : ('a -> 'a)
```

The continuation passing style analogue is a command with two arguments (the input and the output):

```
#let id = (x,y) -> y x;;
Value id = <com> : ('a * ~'a)
```

Let us see now a simple example of recursive definition. For that purpose we make a type definition:

```
#type 'a list = Cons of 'a * 'a list | Nil;;
```

Here is the program for concatenation of lists:

```
#let rec append = function (Cons(x,X),Y) -> Cons(x,append(X,Y))
#                        | (Nil,Y)       -> Y;;
Value append = <fun> : ('a list * 'a list -> 'a list)
```

and its analogue:

```
#let rec append = (Cons(x,X),Y,Z) -> append(X,Y,U -> Z(Cons(x,U)))
#                | (Nil,Y,Z)       -> Z Y;;
Value append = <com> : ~('a list * 'a list * ~('a list))
```

To illustrate side effects, we consider a program which applies a program for printing elements to each element of a list:

```
#let print_list = function (print,X) -> printrec X
#
# where rec printrec = function Cons(y,Y) -> print y; printrec Y
#                     | Nil       -> ();;
Value print_list = <fun> : (('a -> 'b) * 'a list -> unit)
```

and its analogue:

```
#let print_list = (print,X,f) -> printrec(X,f)
#
# where rec printrec = (Cons(y,Y),f) -> print(y,() -> printrec(Y,f))
#                    | (Nil,f)       -> f();;
Value print_list = <com> : ~(~('a * ~unit) * 'a list * ~unit)
```

Finally, we illustrate exceptions with an algorithm testing membership in a tree:

```
#type 'a tree = Node of 'a tree * 'a tree | Leaf of 'a;;

#let member = function (x,X) -> exception success in
#                              (try memrec X; false with success -> true)
#
# where rec memrec = function Node(Y,Z) -> memrec Y; memrec Z
#                           | Leaf y    -> if x=y then raise success;;
Value member = <fun> : ('a * 'a tree -> bool)
```

Its analogue is not too bad:

```
#let member = (x,X,b) -> memrec(X,() -> b false)
#
# where rec memrec = (Node(Y,Z),f) -> memrec(Y,() -> memrec(Z,f))
#                  | (Leaf y,f)    -> if x=y then b true else f();;
Value member = <com> : ~('a * 'a tree * ~bool)
```

References

[CAML] G. Cousineau & G. Huet, *The CAML Primer*, Rapport Technique INRIA 122 (September 90).

[Fisher72] M. J. Fischer, Lambda calculus schemata, in *Proc. ACM Conference on Proving Assertions About Programs*, p 104-109 (1972).

[Gallier91] J. Gallier, *Constructive Logics. Part I: A tutorial on Proof Systems and Typed λ-Calculi*, Research Report Digital PRL 8 (May 1991).

[Girard91] J.Y. Girard, A new constructive logic: classical logic, to appear in *Mathematical Structures in Computer Science* (Cambridge University Press).

Experience with FS_0 as a Framework Theory

Sean Matthews Alan Smaill David Basin*
Edinburgh

July 1, 1991

Abstract

Feferman has proposed a logic, FS_0, as an alternative system for encoding logics and also for reasoning about those encodings. We have implemented a version of this system and performed experiments that show that his system is practical. Specifically, we describe a formalization of predicate calculus and the development of an admissible rule that manipulates formulas with bound variables. This application will be of interest to researchers working with frameworks that use mechanisms such as substitution in the lambda calculus to implement variable binding and substitution in the declared logic directly. We suggest that formalizing, and reasoning about, bound variables is not as difficult as is often supposed, and leads to a more powerful theory about the encoded logic.

1 Introduction: Why Metamathematics?

A logical framework is a formal theory that is designed for the purpose of describing other formal theories in a uniform way, and for making the work of building proofs of theorems in those theories easy. Given a description T of a theory in a framework, a proof of a proposition A in that theory becomes a proof of a proposition in the framework theory that the encoding of A has the property 'provable in T'. But a proof in an encoded theory, when described like this, looks very like a meta-theorem, and a natural question is: 'can we prove more powerful theorems in the framework theory, that say general things about a particular encoded theory?'. To which the answer is 'yes', if the framework theory has certain facilities.

If it is possible to do metamathematics in a framework, then there are at least two reasons for actually doing it. The first is the simple reason that it is interesting in the same way as doing mathematics is interesting, but this paper mostly considers another, more pragmatic, reason, which is that exploiting metamathematics is a way to increase safely the power and usability of theorem proving systems in practice [7].

1.1 The Deduction Theorem

A good example of the sort of advantage that formal metamathematics can offer is the deduction theorem [15] for a logic presented in a Hilbert style. If the notation $\vdash_T A$ means that A is provable in the object-theory T, and $T[B]$ is the theory T extended with the axiom B, then the deduction theorem says

*Mathematical Reasoning Group, Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh. {sean,smaill,basin}@ai.ed.ac.uk

If it is possible to show $\vdash_{T[B]} A$, then it is possible to show $\vdash_T B \to A$.

This is an interesting theorem for several reasons. First, it is an explicitly meta-level result, relating a theory to an extended version of itself; it cannot be stated for theory T inside theory T in a way so that it can be used there. Secondly, it is a perfect example of a meta-theorem that makes work in the object-theory easier — the deduction theorem is often the first theorem proven about a logic (either that, or there is a proof that it does not hold). Thirdly, proofs of theorems in the object-logic can be built in fewer steps by exploiting the meta-theorem. We *know* that the object level proof can be built if it is really needed, and that is enough; we do not usually have to go to the trouble of actually building it.

But usually there is no way of sharing the sort of knowledge embodied in the deduction theorem properly with a proof development system (PDS) for T, since it will not have been supplied with facilities that allow it to make use of such meta-level results. The most that is usually possible, if the PDS has a tactic facility, is for the user to inspect the proof of the meta-theorem and build a tactic that does the same thing (since here the proof amounts to the description and verification of an algorithm — a tactic — that takes a proof of $\vdash_{T[B]} A$ and transforms it into a proof of $\vdash_T B \to A$). But this is not really satisfactory: the deduction theorem justifies a simple constant time syntactic transformation on an object-level goal, but a corresponding tactic will take time dependent on the size of the proof supporting that goal.

To be fair, the deduction theorem is a slightly artificial example, if only because it is such a useful result that the logic of any PDS that is usable in practice is presented in a way (sequent calculus or natural deduction) that provides it as a basic rule. But it illustrates very effectively the gains that meta-level reasoning can provide.

1.2 Possible Advantages and Disadvantages

The most obvious advantage of meta-theoretic extension then, is the speed up that it can bring to practical theorem proving. The gain is not always as dramatic as it is above, in the deduction theorem, but something is usually possible.

Another advantage is less obvious, but at least as important. Tactics are informal, they have no associated formal specifications which describe what they are supposed to do, and which they can be compared against. Instead, what they are expected to do is an implicit side effect of the proof manipulations that they actually describe. For instance, it would probably not be immediately clear from reading the undocumented code of a tactic implementing the deduction theorem, that the effect of running it would be the effect that the deduction theorem describes. The problem with informal tactics is even worse than the problem with most informally specified code, since the sort of operation that a sophisticated tactic performs is much more subtle than typical program code, and so will have correspondingly subtle bugs, which in turn are much more difficult to trace and fix. Further, as tactics are combined together to build more powerful tactics, holding on to any intuition for how they behave becomes more difficult. To remedy this, it is easy to imagine a system for verifying a tactic formally against pre-, and post-, conditions which would capture the intended behaviour, and make it much easier to keep control of what the set of tactics is supposed to do as it grows in size[1]. It is easy to imagine such a facility, but it is difficult to imagine why one would be built. If a tactic has been verified against a transformation described by pre- and post- conditions, why run it at all? Why not just check

[1] In fact, formal pre- and post- conditions themselves could be exploited in various ways (such as combining tactics by reasoning about the conditions, as Bundy suggests in [6]).

that the pre- condition holds, then perform the transformation described by the post- condition directly? Once this obvious step is taken, the system allows the sort of meta-level reasoning that is discussed above.

There are disadvantages too in extending the system's strength by proving meta-theorems rather than by verifying tactics. In the same way as tactics can be subtle, difficult pieces of code, meta-level proofs can also, sometimes, be subtle and difficult to construct. It can be difficult to find a formalisation for the meta-theory of a typical logical system that is comfortable to work in. In particular, there are often supposed to be problems with the formalisation of bound variables. Even without such specific problems, it usually takes longer to construct a formally verified program (which is what a meta-theorem resembles) than it does to construct the same program informally. So a meta-theorem is not always going to best a tactic. But when a tactic has a big complexity disadvantage compared to the implementation of a meta-theorem, or tries to do something that is very subtle, or if it is used often, then there is a good case for replacing it.

A second disadvantage of using a meta-theorem is not always so important: the speed up it gives is because it does not actually construct an object-level proof, all it does is guarantee that such a proof could be constructed. But there are times when the actual proof is needed: for instance if the witness term of a constructive proof is needed, or a transformation tactic (i.e., a tactic from proofs to proofs, instead of from sequents to sequents) is to be applied, then it is not enough simply to know that the proof exists. One possible solution to this would be if the the meta-theory is constructive (e.g., as proposed in [3]), then the actual proof could be reconstructed by examining the proof of the meta-theorem.

In conclusion then, if we are interested in provability, rather than in building object-level proofs, and if we are prepared to do the work of verifying meta-level assertions, then the use of theorem proving in a framework theory gives us a powerful and secure way to extend our capability to obtain results for a declared theory.

1.3 Overview of Paper and Research Contributions

In the remainder of this paper we will look at what is needed from a logical framework if it is to be practically useful for formulating, proving, and using meta-theorems. In §2 we motivate and present Feferman's system FS_0 [10] which we have implemented and in which we have performed a number of experiments. FS_0 is a theory of functions and classes of expressions embedded in second-order predicate calculus. The syntax and theories of formal systems are presented in FS_0 using *finitary inductive definitions.* To illustrate, we formalize a fragment of first-order predicate calculus. In §3 and §4 we provide details on the practical aspects of doing metamathematics in FS_0. In particular, we present an example of a specific meta-theorem that we have proved: a prenex form theorem for the formalized fragment of predicate calculus. We conclude by drawing lessons from our experience and suggesting directions for future work.

Our emphasis throughout is on the practical side of meta-theory. In [10], Feferman lays out FS_0 as a theoretical framework for encoding and reasoning about logics. But, as he himself points out there, in the conclusion: 'whether implementation is feasible and what its value might be can only be judged by actually trying to carry it out'. We have done just that and we think our experiments (only one of which is reported here, others are summarized in [16]) show that FS_0 in practice does provide a suitable basis for formalizing and using meta-theory. As far as we know[2] ours is the first implementation and use of the system.

[2]and as far as Feferman knows [9]

More generally, we believe that our research provides further support to the claim that meta-theoretic extensibility is practical and desirable. Notable examples of previous research in this area[3] that come to mind are the work of Aiello and Weyhrauch on algebraic simplification in *FOL* [1], the work of Constable and Howe on term matching, rewriting, and simplification in *Nuprl* [14, 13], and the work of Boyer and Moore on metafunctions in their theorem proving system [5]. But there has been little other experimental work in this area. One reason, we suspect, is that meta-theoretic reasoning has a reputation for being difficult. For example, most other logical frameworks are based on type theory or some kind of higher-order logic in which variable binding and substitution are intended to be captured by lambda-abstraction and beta-reduction because it is believed to be difficult to encode 'higher-order syntax,' and operations on this syntax, explicitly. Our work indicates that this is not so. Our formalization of predicate calculus, and the prenex normal form theorem, involved formalizing notions such as free and bound variables, alpha-convertibility, substitution, and the like. These definitions were not difficult, and we needed only to restate definitions found in standard logic texts as explicit inductive definitions. The resulting admissible inference rule can be applied a lot more efficiently than an equivalent tactic, and an application of the rule is itself the first step to skolemisation and various normal forms, and, from there, to unification procedures and other mechanised proof procedures.

Note, by the way, that the alternative approach used in logical frameworks like the *ELF* [13], *Isabelle* [17], and *Lambda-Prolog* [12], of identifying bound variables and substitution with syntax and operations in the framework theory leads to a different style of thinking of, and reasoning about, encoded theories. In this approach, bound variables must be understood in terms of properties of the meta-logic, leading to a less straightforward encoding. Theorems such as that which we present in this paper (when they are possible at all) would have to be understood similarly — the side conditions which we explicitly formalise would have to be treated implicitly.

2 Logical Frameworks for Metamathematics

The notion of formal metamathematics seems, at first sight, to be contained in the notion of a logical framework. But in practice things are not that simple, because what might be useful for working in the declared theory can conflict with what might be useful in the framework theory.

In most of the frameworks that have been suggested, a conscious decision has been made to favour work in the declared theory. For instance, in the *ELF*, which is based on a typed lambda calculus, the substitution mechanism available has been elegantly exploited to make substitution in declared logics easier to deal with; but this same facility interferes badly with attempts to do certain types of metamathematics, because variables in the declared level are also variables in the framework level, and so cannot be easily reasoned about at that level either.

A minimal demand of a framework theory is it should be possible to work comfortably in declared theories — after all that is the reason for having one in the first place — but we also like to be able to prove interesting theorems about the declared logic. This is another area where the *ELF* presents problems: it is a subset of the theory *CC* [8], which is extremely powerful, and in which it is possible to prove very general statements. But the *ELF* itself

[3]We are excluding research in theorem proving (some of it quite exceptional such as Shankar's [19] and Berardi's [4]) in which the theorems proved happen to be of a metamathematical nature, but are not be used as theorem proving procedures in part of a formalized meta-theory.

has been carefully weakened so that it is very difficult to prove these general statements; this has been done to make it easier to prove that the encoding of a declared logic is accurate. Unfortunately it also means that proving general statements in the meta-theory, as opposed to proving specific statements that correspond to particular theorems in the declared theory, is difficult. An explicit facility in the theory for constructing evaluable wfunctions, which could then be used to exploit meta-level results directly, inside the theory, would also obviously be useful.

The points above give a fairly good notion of what we might be looking for in a prospective framework theory. But before suggesting a particular possibility that goes at least some of the way towards meeting the criteria above, it is useful to ask the question: 'what is a formal system?'.

2.1 The theory FS_0

One answer to the question 'what is a formal system?' was supplied by Post [18] and Smullyan [20], who looked at classes of strings closed under sets of inductive definitions as the generalised notion — in effect, the theory of recursively enumerable classes. This view is very effective, but it suffers, in the versions presented there, from the problem of being completely unusable in practice — when practice here means implementing it on a computer, and using it actually to prove theorems in a declared theory. Thus the approach has been ignored as a possible basis for a practical framework until recently. In the last few years however, a new proposal in the same vein has been made by Feferman [10], as a competitor to the type theory school of frameworks.

Feferman's theory differs from those of Post and Smullyan most importantly in its basic data structure: strings have been abandoned in favour of s-expressions. He has been able to learn from the thirty years of experience with the *Lisp* programming language that has been accumulated by programmers. *Lisp* is based on s-expressions, and is still, even after thirty years, one of the most popular languages for symbolic computing. This is a strong argument in favour of the powerful utility of such a data structure in practice[4]. He has embedded this theory in a predicate calculus, and extended it with primitive recursive functions on, and recursively enumerable classes of, s-expressions. The most important feature of the system is the ability to represent finitary inductive definitions via these recursively enumerable classes.

A more formal description of the system is as follows. The system consists of s-expressions, functions on s-expressions, functionals for combining functions, and classes. There are also sorted variables over s-expressions, functions and classes. The whole is embedded in the second order predicate calculus (with quantification over classes and functions), with equality and membership relations.

In what follows we use the following notational conventions. v, w, x, y, z are variables in FS_0 over s-expressions, f, g, h are variables over functions, and X, Y, Z are variables over classes of s-expressions. Also notice that universal quantification of variables that appear free in the statements of theorems or axioms is implicit. Further, U, V are meta-level variables over formulae and F is a meta-level variable over constant functions in FS_0.

First there is the type of s-expressions, **S**. There is only only one explicitly defined s-expression, which is 0 (but, below, we show how others are created out of variables and function applications).

[4]Curiously, the other feature of *Lisp* (the part that appeals to theorists rather than practitioners) that is most often mentioned, is its inspiration by the lambda calculus, which is part of the foundation of several other frameworks.

There are versions of the usual functions (all of type $\mathbf{F} \equiv \mathbf{S} \to \mathbf{S}$) that occur in a primitive recursive system as follows. The comma itself is a function, for pairing, which is defined together with the projection functions π_1 and π_2, where

$$(x, y) \neq 0$$

and

$$\pi_i(x_1, x_2) = x_i$$

which together allow arbitrary s-expressions made up of variables and 0s to be put together, so

$$0, \ldots, (0, (0, 0)), \ldots, ((x, 0), ((x, 0), y)), \ldots,$$

are all s-expressions. Then there are the identity ι, where

$$\iota x = x$$

the constants κ_x, where

$$\kappa_x y = x$$

and the compare function δ, where

$$v = (w, x, y, z) \to \begin{cases} \delta v = y & \text{if } w = x \\ \delta v = z & \text{if } w \neq x \end{cases}$$

$$\neg \exists w \, \exists x \, \exists y, \exists z (v = (w, x, y, z)) \to \delta v = 0$$

(the last of these deals with the case where δ is applied to a 'not well formed' argument, i.e., one that cannot be resolved into a quadruple). Notice that, since all functions take only one argument, function application is usually denoted simply by juxtaposing the name of the function and the argument, though sometimes brackets will be used for clarity.

Function applications to ground s-expressions can then be reduced to normal forms in the usual way.

For convenience, in the rest of this paper the comma is taken as associating to the left, so that (x, y, z) is equivalent to $((x, y), z)$ rather than $(x, (y, z))$, and the composition of two projections $\pi_i(\pi_j x)$ is written $\pi_{ij} x$.

These basic functions can be combined using the three second order combinators $\mathcal{C}$, $\mathcal{P}$ and $\mathcal{R}$, which are all of type $\mathbf{F} \times \mathbf{F} \to \mathbf{F}$. The first two of these are composition and pairing of functions, so that

$$\begin{aligned} \mathcal{C}[f, g]x &= f(gx) \\ \mathcal{P}[f, g]x &= (fx, gx) \end{aligned}$$

The third is the combinator for structural recursion on s-expressions, and is more complicated. It expects to be applied to a pair (x, y), where x is a collection of parameters for the recursion, that remains constant as the function descends through the s-expression, and y is the s-expression that the recursion is upon. So the base and step cases are,

$$\begin{aligned} \mathcal{R}[f, g](a, 0) &= fa \\ \mathcal{R}[f, g](a, (b, c)) &= g(a, b, c, \mathcal{R}[f, g](a, b), \mathcal{R}[f, g](a, c)) \\ \mathcal{R}[f, g]0 &= 0 \end{aligned}$$

(as with the definition of δ, the last of these deals with case when the argument to the function is 'not well formed'). Notice that the combinators are second order only, so they cannot be applied to themselves, i.e., $\mathcal{C}[\mathcal{C}, \mathcal{R}]$ is not well defined.

Again, for the sake of readability, $\mathcal{P}[f, g, h] \equiv \mathcal{P}[\mathcal{P}[f, g], h]$ by analogy with the behaviour of the comma for s-expressions, and $\mathcal{C}[f, g, h] \equiv \mathcal{C}[\mathcal{C}[f, g], h]$

At this point, what has been defined is very similar to pure Lisp restricted to primitive recursive functions: functions on s-expressions can be defined and evaluated, but not reasoned about beyond that. Even so, there is enough here already so that any well formed formula in the language is logically equivalent to one with only one free variable. For example, if $U\langle x, y\rangle$ is a formula with x and y free, then $\vdash_{FS_0} U\langle x, y\rangle$ iff $\vdash_{FS_0} U\langle \pi_1 x, \pi_2 x\rangle$, which has one less free variable.

Now the theory is extended with classes of s-expressions ($\mathbf{C}$), a membership relation $\in$, and various class constructor operations. The first of these is the base class $\{0\}$, where

$$x \in \{0\} \leftrightarrow x = 0$$

There are also the intersection operations $\cup$ and $\cap$ of type $(\mathbf{C} \times \mathbf{C} \to \mathbf{C})$ which have their obvious axioms

$$\begin{aligned} x \in X \cup Y &\leftrightarrow x \in X \vee x \in Y \\ x \in X \cap Y &\leftrightarrow x \in X \wedge x \in Y \end{aligned}$$

and the function inverse operation $\cdot^{-1}\cdot$ of type $\mathbf{F} \times \mathbf{C} \to \mathbf{C}$ where

$$x \in f^{-1}X \leftrightarrow fx \in X$$

These operations are enough between them to provide comprehension for open formulae constructed from conjunctions, disjunctions and equality between terms of type $\mathbf{S}$, in FS_0 as follows. Given an appropriate formula U, convert it into a formula with only one free variable over s-expressions, x. While possible, replace some term of the form (fx, gx) with $\mathcal{P}[f, g]x$ or some term of the form $f(gx)$ with $\mathcal{C}[f, g]x$. Then replace each of the equalities of the form $fx = gx$ with $\mathcal{P}[f, g, \kappa_0, \kappa_{(0,0)}]x = 0$ and then all equalities of the form $fx = 0$ or $0 = fx$ with $x \in f^{-1}\{0\}$. Finally, repeatedly replace some term of the form $x \in X \vee x \in Y$ with $x \in X \cup Y$ or some term of the form $x \in X \wedge Y$ with $x \in X \cap Y$. This will result in a formula U^* of the form $x \in X$ where $\vdash_{FS_0} U^*$ iff $\vdash_{FS_0} U$.

This leaves only the constructor for recursively enumerable classes $\mathcal{I}_2$ (of type $\mathbf{C} \times \mathbf{C} \to \mathbf{C}$) to be defined. This allows us to describe in FS_0 any class Z, which can be informally defined as a base class X, closed under a two place rule

$$\frac{x \in Z \qquad y \in Z}{z \in Z} U(z, x, y)$$

(where $U(z, x, y)$ is an open formula in FS_0 with free variables x, y and z). Formally, if Y is a class such that $(z, x, y) \in Y \leftrightarrow U(z, x, y)$, then $Z \equiv \mathcal{I}_2(X, Y)$ and

$$\begin{aligned} x \in X &\rightarrow x \in Z \\ \exists x\, \exists y (x, y \in Z \wedge (z, x, y) \in Y) &\rightarrow z \in Z \end{aligned}$$

(which, with the induction rules described below, gives comprehension over Σ^0_1 relations, or the recursively enumerable classes). FS_0 only explicitly allows inductive definitions with two dependents, but this is all that is needed (as we explain in §2.2).

Now enough machinery has been described so that it is possible to do the same sort of object-level theorem proving as is possible in systems like *ELF*, though without the extra facilities that make *ELF* easy to use. An encoding of a language L as s-expressions can be defined in FS_0, then, using the inductive definition of L, and theories T for L can be defined in the same way.

But it is not yet possible to prove the sorts of generalised statments about defined theories which were mentioned earlier, which is what we actually want to do. To do this we need to be able to make general statements about classes, and for this purpose there are also two induction axioms: induction over inductively defined classes, and induction over s-expressions. Induction over inductively defined classes is used to show that the defined class is a subset of some other class, i.e., (if $X \subset Y$ is an abbreviation for $\forall x(x \in X \rightarrow x \in Y)$, where x does not occur free in X or Y)

$$X \subset Z \rightarrow \forall w\,\forall x\,\forall y(x \in Z \rightarrow y \in Z \rightarrow (w,x,y) \in Y \rightarrow w \in Z) \rightarrow \mathcal{I}_2(X,Y) \subset Z$$

And induction over the s-expressions is just

$$0 \in X \rightarrow \forall x\,\forall y(x \in X \rightarrow y \in X \rightarrow (x,y) \in X) \rightarrow \forall x(x \in X)$$

2.2 How FS_0 is used

While recursively enumerable classes are theoretically enough to describe any formal system, it is not at all clear that they provide enough so that describing and using a theory is possible in practice. To give some idea of how FS_0 does work, below is a part of the definition of a fragment of the language of first-order predicate logic.

We introduce further notational conventions at this point to allow us to discuss the declared language. We take A, B, C to vary over formulae in the language of first-order predicate logic, and a, b to vary over variables in that language, and Q to vary over quantifiers in that language.

In this encoding the components of the language are all labeled with a constant in the left hand part of the representing s-expression so that they can be distinguished. For instance, the set of variables is defined as the class of s-expressions which have the constant "*var*" (which in turn is defined as an s-expression) in the left hand part, where anything can be placed in the right hand part, so the definition can be thought of as the class

$$var \equiv \{(\text{“}var\text{”}, x) | x \in S\}$$

which, in the proper language of FS_0, is the class comprehending all instances y for which the the formula

$$\pi_1 y = \text{“}var\text{”}$$

is provable. I.e., all instances of the pair ("*var*", x) where where x is not constrained.

Then, if the class of atomic predicates *ap* has been defined already, it is possible to give a definition of the class of well formed formulae in the language of predicate logic where the only propositional connectives are disjunction and negation. First, new distinct constants have to be defined for labeling the various components: "$\forall$", "$\exists$", "$\vee$" and "$\neg$". Then the rule defining the language can be given as the inductive definition:

$$wff(''A'') \quad \longleftarrow \quad ap(''A'')$$

$$\begin{array}{rcl} wff(''A \vee B'') & \longleftarrow & wff(''A'') \wedge wff(''B'') \\ wff(''\neg A'') & \longleftarrow & wff(''A'') \\ wff(''QaA'') & \longleftarrow & wff(''A'') \wedge var(''a'') \end{array}$$

where the arrows indicate an inductive definition (see §2.3); so, for instance, the first line says that A is in *wff* if A is an atomic predicate (i.e., in *ap*).

This can be formalised, in a way suitable for the inductive definition mechanism of FS_0, as follows:

$$\begin{array}{rcl} wff_b & \equiv & ap \\ wff_s^1 & \equiv & \{((\text{“} \vee \text{”}, (x,y)), x, y) | x, y \in \mathbf{S}\} \\ wff_s^2 & \equiv & \{((\text{“}\neg\text{”}, x), x, y) | x, y \in \mathbf{S}\} \\ wff_s^3 & \equiv & \{((Q, (v,x)), x, y) | v \in var, Q \in \{\text{“}\exists\text{”}, \text{“}\forall\text{”}\}, x, y \in \mathbf{S}\} \end{array}$$

(where the $_b$ and $_s$ subscripts indicate base and step cases — notice that only wff_s^1 uses the y in the inductive definition; in the others it is ignored). The class of formulae is the closure under the class of atomic predicates of these definitions and can be written simply as $wff \equiv \mathcal{I}_2(wff_b, wff_s^1 \cup wff_s^2 \cup wff_s^3)$.

Then the encoding of the statement $\forall aA \vee \neg(B \vee C)$ is a member of the class *wff* and is represented by the s-expression

$$(\text{“} \vee \text{”}, (\text{“}\forall\text{”}, (''a'', ''A'')), (\text{“}\neg\text{”}, (\text{“} \vee \text{”}, (''B'', ''C''))))$$

Along side this it is possible to give an abstract definition of the notion of, for instance, a sequent calculus presentation of a theory. A sequent (in this case with an single consequent) can be defined simply as a list of encoded formulae, so that the sequent $A_1, \ldots, A_n \vdash A$ can be encoded as $(0, A_1, \ldots, A_n, A)$ (the 0 on the left marks the end of the list, and corresponds to the 'null' in Lisp).

Then a rule, which, in a sequent calculus, is of the form:

$$\frac{\Gamma_1 \vdash A_1, \ldots \Gamma_n \vdash A_n}{\Gamma \vdash A}$$

can be defined as a relation between a list of sequents and a sequent, which van be conveniently represented as a non-empty list of sequents, the head being the goal, and the tail the subgoals. So if the base class of sequents, $A \vdash A$, is defined

$$base \equiv \{(0, x, x) | x \in wff\}$$

then a schema for sequent calculus presentations in the language *wff* is the closure of *base* under whatever the rules are. So it can be written, in schematic form

$$SC\langle RS\rangle \equiv \mathcal{I}_2(base, RS)$$

(where $SC\langle RS\rangle$ is a piece of syntactic sugar that abbreviates for the expression on the the right and RS is the set of rules for the theory). Then a presentation of first-order predicate logic using the language *wff* is a matter of defining an appropriate set of rules R^{PK} and then defining:

$$PK \equiv SC\langle R^{PK}\rangle$$

Now a proposition A is provable in first-order predicate logic iff the encoding of $\vdash A$ (the sequent with an empty context and the consequent A) is in PK. This definition of a schema for sequent calculus presentations shows why a constructor for inductive definitions with more than two dependents is not needed: the inductive definition of lists of objects requires only two dependents, and then inductive definitions dependent on arbitrary numbers of dependents can be defined in terms of inductive definitions which are dependent on a single list.

The definition of a rule can itself depend on complex inductive definitions, for instance the rule for right existential elimination depends on the definition of substitution, i.e.,

$$\frac{\Gamma \vdash A[t/a]}{\Gamma \vdash \exists a A}\exists\text{–R}$$

would be defined in FS_0 as

$$\exists\text{–R} \equiv \{(0,(x,y'),(x,(\text{“}\exists\text{”},(z,y))))|(y',y,v,z) \in sub_C\}$$

(where x, y, y', v, z correspond to $\Gamma, A, A[t/a], t, a$). And then sub is the class of four place relations between the result of a substitution, the formula to be substituted into, a term to be substituted in, and a variable to be substituted for. It is simply another inductively defined class:

$$\begin{array}{rcl}
sub(\text{"}QaA\text{"},\text{"}QaA\text{"},\text{"}t\text{"},\text{"}a\text{"}) & & \\
sub(\text{"}t\text{"},\text{"}a\text{"},\text{"}t\text{"},\text{"}a\text{"}) & & \\
sub(\text{"}a\text{"},\text{"}a\text{"},\text{"}t\text{"},\text{"}b\text{"}) & \longleftarrow & \text{"}a\text{"} \neq \text{"}b\text{"} \\
& \vdots & \\
sub(\text{"}QbA'\text{"},\text{"}QbA\text{"},\text{"}t\text{"},\text{"}a\text{"}) & \longleftarrow & \text{"}b\text{"} \neq \text{"}a\text{"} \wedge \text{"}b\text{"}\ \textit{not free in}\ \text{"}t\text{"} \wedge \\
& & sub(\text{"}A'\text{"},\text{"}A\text{"},\text{"}t\text{"},\text{"}a\text{"}) \\
sub(\text{"}Qb'A'\text{"},\text{"}QbA\text{"},\text{"}t\text{"},\text{"}a\text{"}) & \longleftarrow & \text{"}b\text{"} \neq \text{"}a\text{"} \wedge \\
& & \text{"}b\text{"}\ \textit{free in}\ \text{"}t\text{"} \wedge \\
& & \text{"}b'\text{"}\ \textit{not free in}\ \text{"}t, A\text{"} \wedge \\
& & sub(\text{"}A''\text{"},\text{"}A\text{"},\text{"}b'\text{"},\text{"}b\text{"}) \wedge \\
& & sub(\text{"}A'\text{"},\text{"}A''\text{"},\text{"}t\text{"},\text{"}a\text{"}) \\
& \vdots &
\end{array}$$

(notice that the definition renames variables to avoid capture — the complete definition can be found in any textbook). There is no dispute that this approach to substitution is not as convenient as what is used in other frameworks, but it is certainly usable, and it has some advantages, which we are able to exploit below.

It is easy to see that membership of ground s-expressions in sub is decidable, and it is easy to construct a tactic that carries out the appropriate derivation automatically[5]. On top of this it is easy to see that a rule application is also a decidable step.

Before moving on, we mention some syntactic sugar that hides the raw representations used in FS_0 for the sake of readability. So instead of writing the encoding of '$\Gamma \vdash A$ is provable in T' in FS_0 as $(\text{"}\Gamma\text{"},\text{"}A\text{"}) \in T$, we will write, simply, $\text{"}\Gamma \vdash A\text{"} \in T$.

[5]It would be possible to provide a function F that satisfied the definition of sub so that $(F(x,y,w),x,y,w) \in sub$, which would be a one step operation that did not need to invoke a tactic at all, but such a development is beyond the scope of this paper, though it is described in [16]

2.3 What We Have Implemented

In the description above, there is little apparently in common between the system that is described as FS_0, and what is used to show FS_0 in use, though hopefully it should be clear that the sugared version does not do anything that is not possible in the original syntax.

In fact, in order to make FS_0 usable in practice, we have implemented a tactic based theorem prover for a sequent style presentation, inside a Prolog interpreter. We have also implemented various facilities that translate between a version of the notation used above and the 'raw' form of FS_0. So we can automatically find solutions to specifications written in the form

$$\{S(x_1, \ldots, x_n) | U(x_n, \ldots, x_n)\}$$

(where S and U are an s-expression and a formula in FS_0, containing free variables $x_1, \ldots x_n$). And also, in the same manner, inductive definitions made up of clauses of the form

$$S(x_1, \ldots x_n) \longleftarrow U(x_1, \ldots x_i)$$

(with certain restrictions because only inductive definitions with two dependents are explicitly allowed). The access to Prolog as a tactic language means, too, that a lot of the work described in the rest of this paper can be heavily automated.

In use the system looks as follows. When the system is used for proving theorems in FS_0, the display looks like:

```
incomplete autotactic(idtac)
v6. (((0,(p2 of v1,p1 of v1),implies),sequent),provablesequent
    in proofsandlistsC)
==> \/(
  (v20
    :: ((v20,proof)in proofsandlistsC)#
       ((p2 of v20)
         =
           (sequent,
             (0,
             (implies,
                (implies,(conj,(pn(1,1,1,1,2)of v2,
                                pn(1,1,2)of v1)),
                         pn(2,1,2)of v1),
                (implies,(pn(1,1,1)of v2,p1 of v1)))))))

by _
```

which shows a step in the construction of a meta-theorem. The goal states the existence of a proof object for a particular, schematic, object level goal, and the hypothesis `v6` is an induction hypothesis. The user can also see the object level schematic sequent, in the language of the declared logic, as follows:

```
[] >> (*object*(pn(1,1,1,1,2)of v2) &
       *object*(pn(1,1,2)of v1)) =>
          *object*(pn(2,1,2)of v1) =>
              *object*(pn(1,1,1)of v2)=>
                  *object*(p1 of v1)
```

which, in terms of the language used in this paper, is the goal:

$$'' \vdash (\pi_{11112}x \wedge \pi_{112}y \to \pi_{212}x) \to \pi_{111}y \to \pi_1 x''$$

3 How to do Metamathematics in FS_0

From the description given above, it should be clear that FS_0 theoretically offers quite a lot of facilities for the sort of metamathematical extensions that are discussed in the initial part of this paper, but the practical details have not been discussed, and it is not clear what is possible in practice.

3.1 Implementing an extension

First, recall what was said in the introduction, that it is possible to see a proof of a sequent in a declared language in two different ways, depending on whether the object-level, or the meta-level, characteristics are what is of interest: either as $\Gamma \vdash A$ is a provable sequent in T, or as $\vdash_{FS_0}$ $''\Gamma \vdash A'' \in T$. With the first perspective a user is constrained to work in the declared logic, while with the second he is able to use arbitrary facilities of FS_0 (including what is available in the declared logic). Similarly, there are two ways to look at a deduction step in T; e.g., in the first view it is of the form

$$\frac{\Gamma' \vdash A'}{\Gamma \vdash A} Rule$$

while the second is an instance $DR\langle Rule \rangle$ of the schema

$$DR\langle AR \rangle \equiv (0, ''\Gamma' \vdash A''', ''\Gamma \vdash A'') \in AR \to ''\Gamma' \vdash A''' \in PK \to ''\Gamma \vdash A'' \in PK$$

(AR is a schematic variable that is instantiated to a relation between lists of sequents — in this case, a singleton list — and a sequent). Here this is just a single application of a primitive rule, but there is no reason why AR has to be instantiated to a basic rule of PK; an arbitrary relation AR' can be used, and if that instantiation is provable, the result is a theorem stating that AR' is a rule admissible for PK.

With the appropriate instance of the schema $DR\langle \cdot \rangle$ proven, we no longer have to build a tactic that takes a proof of $\Gamma \vdash A$ and reduces it to $\Gamma' \vdash A$. Instead, all we have to do is build a tactic that, given $''\Gamma \vdash A''$, builds a sequent $''\Gamma' \vdash A'''$ and then goes on to prove that $(''\Gamma' \vdash A''', ''\Gamma \vdash A'') \in AR'$. We no longer have to be worried about constructing the proof that supports this transformation, since we have proven separately that such a proof always exists.

This goes a good part of the way towards our goal, but it is still not quite satisfactory in two ways. First, there is no guarantee that for any particular formula in the domain of AR' there is a corresponding entry in the range (notice that the domain here is on the right, and the range is on the left, unlike the usual convention); secondly, there is still recourse to a tactic. In the ideal there would be no need for this at all. To achieve the ideal, it is necessary to provide a function F in FS_0 so that

$$\{(0, Fx, x) | x \in \mathit{wff}\} \subset AR'$$

upon which $DR\langle AR' \rangle$ is enough to show that $fx \in T \to x \in T$.

The definition of f is not usually immediate. If it were, we could define AR' directly as the class

$$AR' \equiv \{(Fx, x) | x \in \mathit{wff}\}$$

However we prefer to work with an intermediate, relational, definition, which allows us to separate (in the best traditions of software engineering) what we want to do from how we will go about doing it. This is particularly convenient in FS_0 since it is designed to be good at working with recursively enumerable classes, not recursive functions, and its function construction and reasoning facilities are (we have found in practice) trickier, and more frustrating, to use than those for recursively enumerable classes — not least because they are based on primitive recursion. There is also the fact that the specification of AR will probably translate naturally into a recursively enumerable class. The same, translated into a computationally efficient primitive recursive function, may not be so comprehensible.

There will, of course, be cases where, for efficiency's sake, it is better simply to use the tactic language to get the needed transformation, and then verify at run time that the result satisfies the syntactic relation. Again the decision about which approach to take should be made after looking at the practical advantages and disadvantages for the particular case.

3.2 Lemmas For Work in the Meta-level

When constructing meta level proofs, the same sorts of goals appear again and again, particularly, properties of the binding mechanism have to be proven. We have found the following results to be generally very useful tools for dealing with them, though we would not claim that they make up a complete set. The particular ways that they are proven will vary depending on the sort of binding mechanism that is chosen, but the same results seem to be needed, whether we have a standard, or a de Bruijn, style binding mechanism — in our implementation we use a standard binding mechanism.

The first result that is useful is the substitution lemma

$$\Gamma \vdash A \qquad \text{iff} \qquad (\Gamma \vdash A)[a'/a]$$

(where v' is a variable that does not occur free in the context Γ, A).

The second result is a function that is guaranteed to return a variable that is not free in the current context, i.e., a function nfi_F which is a solution to

$$\mathit{nfi}_F(''\Gamma A'') \textit{ not free in } ''\Gamma \vdash A''$$

Because nfi_F is a function that is actually used in practice, not just any function is a satisfactory solution, we need one that is computationally efficient. Our solution to this is

$$\mathit{nfi}_F x \equiv (\text{“}\mathit{var}\text{”}, x)$$

which takes advantage of the way that the class of variables is defined, and provides a function that is constant time (and constant space, if structure sharing is implemented, like in our system), irrespective of the size of the context x. In fact, further than this, since it is often used in conjunction with a substitution function, and since not only is $\mathit{nfi}_F x$ free in x, but it does not occur at all in x, it is possible to define a much simpler, and faster, substitution function Sub

$$''A[\mathit{nfi}_F(A)/a]'' = \mathit{Sub}(\mathit{nfi}_F(''A''), ''a'', ''A'')$$

that does not need to worry about checking for variable capture, or performing alpha conversion.

The third lemma is less ubiquitous, but still very commonly useful. It says that that identity substitution is an identity operation; i.e.,

$$''A[a/a]'' = ''A''$$

As an aside, it is worth pointing out that though this seems totally obvious, it does not hold for some common formal definitions of substitution, such as that given in [2], where all that we are able to count on is that $A = A[a/a]$ up to alpha-congruence. We cannot be sure of syntactic equivalence.

4 Doing Metamathematics in FS_0

Now we present a concrete example of what has been described above in abstract form; the example we have chosen is the definition and construction of a rule for quantifier prenex form for predicate logic. The rule is provided for the theory *PK* defined above, where the only connectives are the quantifiers, disjunction and negation. The prenex form theorem says:

> For any formula A in classical first-order predicate logic *PK* there exists another formula $A^* \equiv Qa_1 \dots Qa_n B$ where each a_i is distinct from the others, each Q is a universal or existential quantifier, B is open, and such that $\vdash A \leftrightarrow A^*$. From this, the admissible rule $\Gamma \vdash A$ iff $\Gamma \vdash A^*$ follows.

4.1 Defining the Relation

The relation between A and a corresponding prenex form can be written as a relation of the form *pnf*(A, A^*), and defined with an inductive definition. The definition here is slightly different from usual, because we are dealing with only a fragment of the language of *PK*, so it is not possible to use DeMorgan's laws for instance. The details are as follows:

$$\begin{array}{rcl}
\mathit{pnf}(''A'', ''A'') & \longleftarrow & \mathit{ap}(''A'') \\
\mathit{pnf}(''QaA'', ''QaA^*'') & \longleftarrow & \mathit{pnf}(''A'', ''A^*'') \\
\mathit{pnf}(''\neg A'', ''B'') & \longleftarrow & \mathit{pnf}(''A'', ''A^*'') \wedge \mathit{pni}(''A^*'', ''B'') \\
\mathit{pnf}(''A \vee B'', ''C'') & \longleftarrow & \mathit{pnf}(''A'', ''A^*'') \wedge \mathit{pnf}(''B'', ''B^*'') \wedge \\
& & \mathit{joinor}(''A^*'', ''B^*'', ''C'')
\end{array}$$

where the subsidiary definitions are:

$$\begin{array}{rcl}
\mathit{pni}(''A \vee B'', ''\neg(A \vee B)'') & & \\
\mathit{pni}(''\neg A'', ''A'') & & \\
\mathit{pni}(''QaA'', ''\overline{Q}aA^*'') & \longleftarrow & \mathit{pni}(''A'', ''A^*'')
\end{array}$$

and

$$\begin{array}{rcl}
\mathit{joinor}(''A'', ''B'', ''A \vee B'') & \longleftarrow & ''A'' \neq ''QaC'' \wedge ''B'' \neq ''QaC'' \\
\mathit{joinor}(''QaA'', ''B'', ''Qa^*C'') & \longleftarrow & \mathit{joinor}(''A[a^*/a]'', ''B'', ''C'') \wedge ''a^*'' \notin ''A'', ''B'' \\
\mathit{joinor}(''B'', ''A'', ''C'') & \longleftarrow & \mathit{joinor}(''A'', ''B'', ''C'')
\end{array}$$

(Where the notation $\overline{Q}$ just means 'the other quantifier than Q', e.g., $\overline{\forall} \equiv \exists$, and $"a" \notin "A"$ means that a does not occur free in A.) These definitions can easily be converted into the definitions of recursively enumerable sets in FS_0, in the same way as *wff* earlier; so (like with *wff*) classes of the form $pnf_b, pnf_s^1, \ldots, joinor_b, \ldots$ etc. will actually be built by the system.

4.2 Verifying the relation *pnf*

Since this is supposed to correspond to an admissible rule, we will want to prove an instance of $DR\langle\cdot\rangle$, but in order to do that eventually, first we show that

$$\vdash_{FS_0} ("A", "A^*") \in pnf \rightarrow " \vdash A \leftrightarrow A^*" \in PK \qquad (*)$$

where the usual syntactic abbreviations for the various connectives hold, i.e.,

$$\begin{aligned} A \leftrightarrow B &\equiv (A \rightarrow B) \wedge (B \rightarrow A) \\ A \wedge B &\equiv \neg(\neg A \vee \neg B) \\ A \rightarrow B &\equiv \neg A \vee B \end{aligned}$$

4.2.1 An Appropriate Form of the Goal

Since *pnf* has been defined by inductive definition, the obvious way to prove $(*)$ is by induction over classes, but this is not yet possible; the goal has first to be slightly rewritten to

$$pnf \subset \mathcal{P}[\kappa_0, iff_F]^{-1} PK \qquad (*')$$

with the definitions:

$$\begin{aligned} iff_F &= \mathcal{C}[and_F, \mathcal{P}[imp_F, \mathcal{C}[imp_F, rev_F]]] \\ and_F &= \mathcal{C}[neg_F, or_F, \mathcal{P}[\mathcal{C}[neg_F, \pi_1], \mathcal{C}[neg_F, \pi_2]]] \\ imp_F &= \mathcal{C}[or_F, \mathcal{P}[\mathcal{C}[neg_F, \pi_1], \pi_2]] \\ or_F &= \mathcal{P}[\kappa_{``\vee"}, \iota] \\ neg_F &= \mathcal{P}[\kappa_{``\neg"}, \iota] \\ rev_F &= \mathcal{P}[\pi_2, \pi_1] \end{aligned}$$

Then it is fairly easy to see, by simplifying the function applications, that $(*)$ if $(*')$. I.e.,

$$pnf \subset \mathcal{P}[\kappa_0, iff_F]^{-1} PK \leftrightarrow \forall x (x \in pnf \rightarrow x \in \mathcal{P}[\kappa_0, iff_F]^{-1} PK)$$

and

$$x \in \mathcal{P}[\kappa_0, iff_F]^{-1} PK \leftrightarrow \mathcal{P}[\kappa_0, iff_F] x \in PK$$

and, from the definitions above,

$$\begin{aligned} &\mathcal{P}[\kappa_0, iff_F] x \\ &\quad = (0, iff_F x) \end{aligned}$$

$$
\begin{aligned}
&= (0, and_F(imp_F x, imp_F(rev_F x))) \\
&= (0, and_F(or_F((neg_F(\pi_1 x), \pi_2 x), (neg_F(\pi_2 x), \pi_1 x)))) \\
&= (0, and_F(''\neg\pi_1 x \vee \pi_2 x'', ''\neg\pi_2 x \vee \pi_1 x'')) \\
&= (0, and_F(''\pi_1 x \rightarrow \pi_2 x'', ''\pi_2 x \rightarrow \pi_1 x'')) \\
&\vdots \\
&= ''\vdash \pi_1 x \leftrightarrow \pi_2 x''
\end{aligned}
$$

Notice that there is a slight abuse here of the conventions we have defined: terms of FS_0 such as variables and their projections appear as part of the notation for encoded sequents of the declared theory. In fact x is an FS_0 variable ranging over pairs of formulae in the declared language. This is how schematic formulae are represented in our system. Fortunately, this reduction does not need to be done by hand, since the machine is able to take handle the messy details.

Then applying induction over classes to $(*')$ generates two subgoals:
the base case

$$\vdash_{FS_0} x \in pnf_b \rightarrow x \in \mathcal{P}[\kappa_0, iff_F]^{-1} PK$$

and the step case

$$
\begin{aligned}
\vdash_{FS_0} x \in \mathcal{P}[\kappa_0, iff_F]^{-1} PK &\rightarrow \\
y \in \mathcal{P}[\kappa_0, iff_F]^{-1} PK &\rightarrow \\
(w, x, y) \in pnf_s &\rightarrow \\
w \in \mathcal{P}[\kappa_0, iff_F]^{-1} PK &
\end{aligned}
$$

4.2.2 The Base Case

The base case is easy; the definition of pnf_b given above is translated automatically into a formula equivalent to $x \in pnf_b \leftrightarrow (\pi_1 x = \pi_2 x \wedge \pi_1 x \in ap)$, so the base case simplifies to

$$\pi_1 x \in ap \vdash_{FS_0} iff_F(\pi_1 x, \pi_1 x) \in PK$$

which in turn, by simplification and rule applications, is provable if (two instances — one for each direction of the 'implication' — of) the goal

$$''\pi_1 x \vdash \pi_1 x'' \in PK$$

is provable. And, since it is a basic sequent, and therefore contained in the base class, *base*, of PK, this follows.

4.2.3 The Step Case

The step case is, naturally, more difficult: here there are three sub-cases, depending on whether the outermost connective is a quantifier, a disjunction or a negation. Assuming that the step case has been reduced to a subgoal for each of these cases, we will work through the first, since it is the easiest and makes all the important points without being obscured with details. The goal that has to be proven is:

$$x, y \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1}PK,\ (w, x, y) \in \mathit{pnf}_s^1 \vdash_{FS_0} w \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1}PK$$

which by thinning, simplification, and application of rules available in R^{PK}, reduces to:

$$\begin{aligned}&{''}\pi_1 x \vdash \pi_2 x{''} \in PK,\\ &\quad {''}\pi_2 x \vdash \pi_1 x{''} \in PK,\\ &\qquad y \in \mathit{var},\\ &\qquad\quad Q \in \{\text{``}\exists\text{''}, \text{``}\forall\text{''}\}, \vdash_{FS_0} ({''}Qy(\pi_1 x){''}, {''}Qy(\pi_2 x){''}) \in \mathcal{P}[\kappa_0, \mathit{iff}_F]^{-1}PK\end{aligned}$$

Again, we will restrict our worked example, this time to where $Q =$ "$\forall$", and to one direction of the iff that needs to be proven, so of the four goals that further simplification produces we will discuss only

$${''}\pi_1 x \vdash \pi_2 x{''}, {''}\pi_2 x \vdash \pi_1 x{''} \in PK,\ y \in \mathit{var} \vdash_{FS_0} {''}\forall y(\pi_1 x) \vdash \forall y(\pi_2 x){''} \in PK$$

The goal here is very similar in form to a goal in the declared theory, as described earlier in this paper, except that it is schematic; x is schematic over pairs of formulae and y is schematic over variables. This does not stop us treating it like an ordinary object-level goal though, so long as we realise that some of the housekeeping that is automatically checked when the goal is unschematic will instead have to be dealt with manually. Then an application of the rule $\forall$–R substituting in the same schematic variable y, will reduce the goal to

$${''}\pi_1 x \vdash \pi_2 x{''}, {''}\pi_2 x \vdash \pi_1 x{''} \in PK,\ y \in \mathit{var} \vdash_{FS_0} {''}\forall y(\pi_1 x) \vdash \pi_2 x{''} \in PK$$

with the side conditions that y is not free in $\forall y(\pi_1 x)$, and $\pi_2 x[y/y] = \pi_2 x$. The first of these can still be checked to be true automatically, since whatever y stands for, it is certainly bound by the outside quantifier of $\forall y(\pi_1 x)$, so the checking procedure does not need to look at $\pi_1 x$ at all. The checker we have implemented cannot deal with the second side condition, but can be demonstrated by appeal to the third lemma of in §3.2. Then an application of $\forall$–L, again using y, produces the goal

$${''}\pi_1 x \vdash \pi_2 x{''}, {''}\pi_2 x \vdash \pi_1 x{''} \in PK,\ y \in \mathit{var} \vdash_{FS_0} {''}\pi_1 x \vdash \pi_2 x{''} \in PK$$

which produces a side condition similar to the second of those above, and is proved in the same way; so we are left with the a goal which is provable from one of the hypotheses.

Essentially the same procedure is followed for the other subgoals, except that it is necessary to prove the lemmas

$$\vdash_{FS_0} x \in \mathit{pni} \to {''} \vdash \neg\pi_1 x \leftrightarrow \pi_2 x{''} \in PK$$

and

$$\vdash_{FS_0} x \in \mathit{joinor} \to {''} \vdash (\pi_{11} x \vee \pi_{21} x) \leftrightarrow \pi_2 x{''} \in PK$$

which are proven in the same manner as $(*)$. It is worth pointing out here that the problem of finding a new bound variable with which to perform alpha conversion in the second branch of the definition of *joinor* does not actually arise, since the induction is able to assume, by the definition of the relation, that such a variable is available.

What is most remarkable about this proof is how much of it, once the initial preparation had been done, can be done in pretty much the same way as an ordinary object-level proof. In fact, as long as some care is taken in their construction, there is no reason why tactics designed for ordinary object level proofs in PK should not in general be usable for meta-level reasoning in PK[6], so that a lot of the proof at the meta-level is not very different from constructing a proof at the object level.

Given the proof of $(*)$ we can now prove the appropriate instance of the $DR\langle\cdot,\cdot\rangle$ schema. First it is necessary to construct an appropriate relation between a list of sequents and a sequent, which can be easily defined as

$$Pnf(''\Gamma \vdash A^*'', ''\Gamma \vdash A'') \quad \longleftarrow \quad pnf(''A'', ''A^*'')$$

then $\vdash_T DR\langle Pnf, PK\rangle$ quickly reduces, after simplification, and cutting in $(*)$, to:

$$''\Gamma_1 \vdash A_1'' \in PK,\ ''\vdash A \leftrightarrow A^*'' \in PK \vdash_{FS_0} ''\Gamma \vdash A^*'' \in PK$$

which again, after shifting perspective to the object level, is a trivial proof.

Now we have achieved the first part of what is described in §3.1, so we could build a tactic that would take any $''\Gamma \vdash A'' \in PK$ construct an appropriate $''\Gamma \vdash A^*''$, and then, after cutting in and eliminating the appropriate instance of $DR\langle Pnf, PK\rangle$, would construct a proof disposing of the resultant subgoal

$$(''\Gamma \vdash A^*'', ''\Gamma \vdash A'') \in Pnf$$

allowing the admissible rule step to be performed after only the syntactic transformation has been verified; the tactic is no longer 'indirect', since it explicitly performs what it is intended to do, i.e., a transformation of the goal into a prenex form.

4.3 Constructing and Verifying the Function

But if we stop here, we are still need a tactic, so we still do not have a 'one step' derivation at the user interface. To fix this, the next part of the development is the construction of a function pnf_F that can be verified to perform the transformation in one step.

The definition of *pnf* itself clearly corresponds to an algorithm, but the algorithm is not encodable directly in FS_0: the only computation facility available directly in FS_0 is primitive recursion on s-expressions. We have implemented a facility that is able to solve specifications for functions that use course of values recursion on s-expressions, and is easily able to deal with functions for *pnf* and *pni* defined

$$\begin{aligned}
pnf_F(''A \vee B'') &= joinor_F(pnf_F(''A''), pnf_F(''B'')) \\
pnf_F(''\neg A'') &= pni_F(pnf_F(''A'')) \\
pnf_F(''QaA'') &= ''Qa(pni_F(''A''))'' \\
pnf_F(''A'') &= ''A'' \\
pni_F(''QaA'') &= ''\overline{Q}a(pni_F(''A''))'' \\
pni_F(''A \vee B'') &= ''\neg(A \vee B)'' \\
pni_F(''\neg A'') &= ''\neg A''
\end{aligned}$$

[6]Strictly, this does not apply to transformation tactics for PK, of course.

(It should be noted that the branches are assumed to be tested in order, so that the last branch of, for instance, pnf_F above, is the 'otherwise' case — it is assumed implicitly that none of the possibilities above it in the list hold). The definition of $joinor_F$ presents greater difficulties though, since its definition is not course of values. But the definition can be rewritten using course of values recursion if a secondary function is used

$$\begin{aligned}
joinor_F(\text{"}QaA\text{"},\text{"}B\text{"}) &= \text{"}Qa^*(joinor_F(Sub(\text{"}a^*\text{"},\text{"}a\text{"},\text{"}A\text{"}),B))\text{"} \\
&\quad \text{where } \text{"}a^*\text{"} = nfi_F(\text{"}A,B\text{"}) \\
joinor_F(\text{"}A\text{"},\text{"}B\text{"}) &= joinor'_F(\text{"}A\text{"},\text{"}B\text{"}) \\
joinor'_F(\text{"}A\text{"},\text{"}QaB\text{"}) &= \text{"}Qa^*(joinor'_F(\text{"}A\text{"},Sub(\text{"}a^*\text{"},\text{"}a\text{"},\text{"}B\text{"})))\text{"} \\
&\quad \text{where } \text{"}a^*\text{"} = nfi_F(\text{"}A,B\text{"}) \\
joinor'_F(\text{"}A\text{"},\text{"}B\text{"}) &= \text{"}A \vee B\text{"}
\end{aligned}$$

We now show how to prove that the relation

$$pnfF \equiv \{(x,y)|x \in wff, y = pnf_F x\}$$

is a subrelation of *pnf*, i.e., that

$$pnfF \subset pnf \qquad (**)$$

This cannot be done in the same way as in the proof of the apparently similar $(*')$ because *pnfF* is not given as an inductively defined class. Since we are trying to verify a function, we want to use induction over s-expressions, which we can do, indirectly, as follows: *pnfF* defines as a primitive recursive class, and therefore its complement $\overline{pnfF}$ is definable in FS_0 as

$$\overline{pnfF} \equiv \{(x,y)|y \neq pnf_F x\}$$

So, since $(**)$ is an obvious consequence of

$$\begin{aligned}
&x \notin pnfF \vee x \in pnf \\
&\quad \leftrightarrow \quad x \in \overline{pnfF} \vee x \in pnf \\
&\quad \leftrightarrow \quad x \in \overline{pnfF} \cup pnf
\end{aligned}$$

and the last of these can be tackled easily by a course of values induction on s-expressions (the path of the proof simply follows the structure of the definition of the relation), the result follows.

Then if *PnfF* is defined as

$$PnfF(\text{"}\Gamma \vdash A^*\text{"},\text{"}\Gamma \vdash A\text{"}) \quad \longleftarrow \quad pnfF(\text{"}A\text{"},\text{"}A^*\text{"})$$

it is easy to show that $DR\langle PnfF\rangle$ implies

$$\text{"}\Gamma \vdash pnf_F(\text{"}A\text{"})\text{"} \in PK \to \text{"}\Gamma \vdash A\text{"} \in PK$$

This is a genuine one step deduction, since the normalisation of $pnf(A)$ can be done by the PDS itself.

At this stage we have constructed a function that transforms a formula in the language of *wff* into its prenex form, in one step, inside FS_0. Furthermore, we have proved formally, in FS_0, that this transformation is admissible in the theory *PK*, i.e., that

$$\text{"}\Gamma \vdash pnf_F(\text{"}A\text{"})\text{"} \in PK \to \text{"}\Gamma \vdash A\text{"} \in PK$$

is valid for *PK*, which is what we wanted to do.

5 Conclusions and Further Work

Above, we have described work showing that Feferman's proposed system FS_0 is a practical option for a framework theory, and further that it is possible to use FS_0 to prove meta-level theorems that can be used to extend a declared logic safely and efficiently. Specifically, we have shown that it is possible to reason about formal theories that use bound variables quite easily.

The total specifications for the language and theory used in the example described here consisted of about a thousand lines of definitions, but a lot of this was basic definitions and the substitution mechanism. It was also the first large definition that has been constructed with the current support facilities. A reconstruction of the same work would, we think, be smaller, and would also be more modular — as the specification is at the moment it is difficult to see which definitions are specific to the definition of PK and which are more general. The specification of (a slightly more sophisticated form of) the normal form theorem consisted of about a hundred lines of code, and the two parts of the proof (showing that the relation is valid, and constructing a function that satisfies the relation) take about a day to construct each.

5.1 Further work, and other considerations

A lot of the work that has to be done in building proofs that deal with bound variables can be disposed of by appeal to a set of lemmas. An examination of these lemmas, and how a change in the binding mechanism affects them, should give insight into what would be needed from a framework that combined the convenient handling of bound variables that is available in *ELF*-style frameworks, with the convenient reasoning about them that is possible in FS_0.

One particular weakness of FS_0 is its lack of proof-theoretic strength (identical to primitive recursive arithmetic). This restricts the sorts of meta-level theorems that can be proven (the problem is particularly significant for meta-level theorems, which often need to appeal to stronger forms of induction than are available in primitive recursive arithmetic). Feferman himself, in another strand of his work, has provided a possible solution to this problem though. He has suggested using reflection principles to strengthen the proof strength of theories. Indeed, the idea that this technique be applied to FS_0 is implicit, if not explicit, in his recent paper [11], that the ideas there be applied to FS_0.

Another direction of research that seems to offer great possibilities that are not available in other frameworks, is the notion of schematic presentations of theories. An examination of the formal proof of the theorem presented in this paper shows that, for instance, wherever the theory $PK \equiv SC\langle R^{PK}\rangle$ is used, it would be easily possible instead to prove a version of theorem, abstracted over a class EX, that used the theory $PK' \equiv SC\langle R^{PK} \cup EX\rangle$, thus proving that the rule is admissible in any extension of PK. But we have only started to consider the implications of this.

References

[1] L. Aiello and R. Weyhrauch. Using meta-theoretic reasoning to do algebra. In *5th International Conference On Automated Deduction*, Les Arcs, France, 1980.

[2] H. Barendregt. *The Lambda Calculus*. Elsevier, 1985.

[3] D. Basin and C. Constable. Metalogical frameworks. In *Proceedings of the Second Workshop on Logical Frameworks*, Edinburgh, Scotland, 1991. To Appear.

[4] S. Berardi. Girard normalization proof in LEGO. In *Proceedings of the Second Workshop on Logical Frameworks*, Edinburgh, Scotland, 1991.

[5] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–184, Academic Press, 1981.

[6] A. Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference On Automated Deduction*, pages 111–120, Argonne, Illinois, 1988.

[7] R. Constable and D. Howe. Implementing metamathematics as an approach to automatic theorem proving. In *A source book of formal approaches to A.I.*, North Holland, 1990. To appear.

[8] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 95–120, 1988.

[9] S. Feferman. Private Communication.

[10] S. Feferman. Finitary inductively presented logics. In *Logic Colloquim '88*, North-Holland, 1989.

[11] S. Feferman. Reflecting on incompleteness. *Journal of Symbolic Logic*, 56:1–49, 1991.

[12] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *9th International Conference On Automated Deduction*, Argonne, Illinois, 1988.

[13] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*, pages 193–204, IEEE, 1987.

[14] D. Howe. Computational metatheory in Nuprl. In *9th International Conference On Automated Deduction*, pages 238–257, Argonne, Illinois, 1988.

[15] S. Kleene. *Introduction to Metamathematics.* Van Nostrand, New York, 1952.

[16] S. Matthews. *Meta Level and Reflexive Extension in Mechanical Theorem Proving.* PhD thesis, University of Edinburgh, DAI, 1991. Expected.

[17] L. Paulson. Natural deduction proof as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1985.

[18] E. Post. Formal reductions of the general combinatorial decision problem. *Amer. J. Math*, 65:197–214, 1943.

[19] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1:407–434, 1985.

[20] R. Smullyan. *Theory of Formal Systems.* Princeton University Press, New York, 1961.

A Mechanized Theory of the π-calculus in HOL

(*extended abstract*)

T. F. Melham
University of Cambridge Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

June 1991

The π-calculus is a process algebra developed by Milner, Parrow and Walker [13, 14] for modelling concurrent systems in which the pattern of interconnection between processes may change over time. This paper describes work in progress on a mechanized formal theory of the π-calculus in higher order logic using the HOL theorem prover. The main aim of this work, which is being done jointly with Mike Gordon at the Computer Laboratory in Cambridge, is to construct a practical and sound theorem-proving tool to support reasoning about applications using the π-calculus as well as metatheoretic reasoning about the π-calculus itself.

Four general principles are adopted in this project. First, a purely definitional approach is taken to embedding the π-calculus in logic. Second, proofs in the π-calculus are automated wherever feasible, with a view to eventually using the system to reason about applications. In practice, this has so far meant writing efficient derived inference rules in HOL for proving decidable classes of propositions such as the α-equivalence of two terms in the calculus. The third principle is to make the HOL proofs as robust as possible, in the sense that they should run without major modification even when minor changes are made to the π-calculus itself. In doing so, we hope to facilitate experimental investigations in HOL of minor variants of the calculus. Finally, the π-calculus as mechanized in HOL is intended to be as nearly identical as possible to the calculus as described in the reports [13, 14]. The aim is to avoid simplifying the calculus merely in order to make the job of mechanizing it easier. One point at which we have compromised this last principle is discussed below in section 4.3.

1 The HOL system

The HOL system [6] is a mechanized proof-assistant developed by Mike Gordon at the University of Cambridge for generating proofs in higher order logic. HOL is based on the LCF approach to interactive theorem proving and has many features in common with the LCF systems developed at Cambridge [15] and Edinburgh [8]. Like LCF, the HOL system supports secure theorem proving by representing its logic in the strongly-typed functional programming language ML. Propositions and theorems of the logic are represented by abstract data types, and interaction with the theorem prover takes place by executing ML procedures that operate on values of these data types. Because HOL is built on top of a general-purpose programming language, the user can write arbitrarily complex programs to implement proof strategies. Furthermore, because of the way the logic is represented in ML, such user-defined proof strategies are guaranteed to perform only valid logical inferences.

1.1 The HOL logic

The version of higher order logic supported by HOL is based on Church's formulation of simple type theory [3]. The HOL logic extends Church's formulation in two significant ways: the syntax of types includes the polymorphic type discipline developed by Milner for the LCF logic PPλ [8], and the primitive basis of the logic includes explicitly-stated rules of definition for consistently extending the logic with new constants and new types. As this second feature of the logic is particularly relevant to the approach taken to embedding the π-calculus in HOL, the rules of definition in the HOL logic are very briefly introduced below. A full description of these rules and details of the rest of the logic, including a set-theoretic semantics, can be found in [4].

1.1.1 Primitive rules of definition

The HOL user community has a very strong tradition of taking a purely definitional approach to using higher order logic, and this is the way in which the logic is used in the present work on the π-calculus. The advantage of this approach, as opposed to the axiomatic method, is that the primitive rules of definition admit only conservative extensions to the logic and are therefore guaranteed to preserve consistency. The disadvantage is that this allows one to make definitions of only certain very restricted forms; all other kinds of definitions must be derived by formal proof from equivalent but possibly rather complex primitive definitions.

The primitive basis of the HOL logic includes three rules of definition: the rule of *constant definition*, the rule of *constant specification*, and the rule of *type definition*. A constant definition is simply an equational axiom $\vdash \mathsf{c} = t$ that introduces a new constant c as an object-language abbreviation for a closed term t. Also admitted by this rule are curried or paired function definitions of the form:

$$\vdash \mathsf{f}\, v_1\ v_2\ \ldots\ v_n = t$$

Among the side-conditions of the rule of constant definition, the details of which are not relevant here, is the condition that the constant being defined may not occur on the right-hand side of its defining equation. This rules out, at least as primitive, all recursive definitions—including inconsistent ones like $\vdash \mathsf{c} = \neg\mathsf{c}$.

The rule of constant specification allows one to introduce a new constant into the logic as an atomic name for a quantity already known to exist. By this rule of definition, one may infer from a theorem of the form $\vdash \exists x.\, P[x]$ a theorem $\vdash P[\mathsf{c}]$, where c is a new constant. This simply introduces c as an object-language name for an existing value x for which $P[x]$ holds.

The third primitive rule of definition in HOL is the rule of type definition. Suppose that σ is a type and $P{:}\sigma{\to}\mathit{bool}$ is the characteristic predicate of some useful nonempty subset of the set denoted by σ. A type definition simply introduces a new type constant t to name this subset of σ. From the theorem $\vdash \exists x{:}\sigma.\, P\ x$, one may infer by the rule of type definition the existence of a bijection from the values of a new type t to the set of values that satisfy P:

$$\vdash \exists f{:}t{\to}\sigma.(\forall x\ y.\, f\ x = f\ y \supset x = y) \wedge (\forall x.\, P\ x = (\exists y.\, x = f\ y))$$

This definitional theorem introduces the new type constant t to name the nonempty set of values whose properties are determined by the choice of predicate P. The requirement that $\vdash \exists x.\, P\ x$

ensures that there is at least one value of type t. This restriction is necessary because the HOL logic does not allow empty types. The rule of type definition can also be used to define new type operators; see [10] for a series of detailed examples.

1.1.2 Derived rules of definition

The primitive rules outlined above disallow the direct use of many commonly-used principles of definition—for example, function definitions by primitive recursion. The general-purpose language ML, however, provides a facility in HOL for supporting fully automatic derived rules of definition. Among the derived principles of definition supported by HOL are recursive concrete type definitions and primitive recursive function definitions over these types, as well as certain forms of inductive definition.

The HOL mechanization of the π-calculus outlined below is a purely definitional theory in higher order logic. It relies heavily on the derived principles of definition available in HOL, which are therefore briefly explained as they used in the sections that follow. Details of these derived rules of definition can be found in the HOL system documentation [4].

2 The π-calculus

This section provides a summary overview of the π-calculus in just enough detail for a reader familiar with (for example) CCS [12] to follow the HOL mechanization described in later sections. For full details of the π-calculus and for motivational discussion, the reader should consult the LFCS technical reports by Milner, Parrow and Walker [13, 14]. The summary presented here is based on the material in these reports, and of course no claim to originality in respect of the ideas in this section is made by the present author.

2.1 Syntax of the calculus

Let $\mathcal{N}$ be an infinite set of *names*, which in the π-calculus are used both as variables and as data values, as well as names of the ports or communication links between processes. The syntax of *agents* in the π-calculus is defined by:

$$
\begin{array}{rcll}
P & ::= & \mathbf{0} & \text{inaction} \\
 & | & \overline{x}y.P & \text{output } y \text{ on } x \text{ then } P \\
 & | & x(y).P & \text{input } z \text{ on } x \text{ then } P\{z/y\} \\
 & | & \tau.P & \text{do silent } \tau \text{ then } P \\
 & | & P_1 + P_2 & P_1 \text{ or } P_2 \\
 & | & P_1 \mid P_2 & P_1 \text{ and } P_2\text{, in parallel} \\
 & | & (x)P & \text{restrict scope of } x \\
 & | & [x{=}y]P & \text{if } x = y \text{ then } P \text{ else } \mathbf{0} \\
 & | & A(x_1, \ldots, x_n) & \text{defined agent}
\end{array}
$$

where P, P_1, P_2 range over agents, x, x_1, $\ldots$, x_n, y range over names, and A ranges over n-ary *agent identifiers*. The forms $x(y).P$ and $(y)P$ introduce variable binding into the calculus; the prefixes '$x(y)$' and '(y)' bind the name y in P. If an occurrence of a name y is not bound, it is called *free*. The set of names that occur free in an agent P is written $\mathrm{fv}(P)$, and the set of names bound in an agent P is written $\mathrm{bv}(P)$. The set of names of an agent P, written $\mathrm{v}(P)$, is defined to be the union of $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$.

Agent identifiers provide the π-calculus with both abbreviations for classes of agents and recursion. Each n-ary agent identifier A is equipped with a defining equation of the form

$$A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P$$

where the set of all names that appear free in P is a subset of $\{x_1, \ldots, x_n\}$. A defining equation or a set of such equations may be recursive, and hence may introduce infinite behaviour.

The meaning of agents is informally and very briefly summarized as follows. The agent $\mathbf{0}$ does nothing. The agent $\overline{x}y.P$ emits the name y on the output port x and then behaves like P. The agent $x(y).P$ receives a name z on the input port x and then behaves like $P\{z/y\}$, where '$P\{z/y\}$' denotes the result of substituting z for every free occurrence of y in P, with change of bound names if necessary to avoid capture of z. The agent $\tau.P$ performs the silent action τ and then behaves like P. As in CCS, the agent $P_1 + P_2$ behaves like either P_1 or P_2, and the agent $P_1 \mid P_2$ represents the parallel composition of P_1 and P_2. In the agent $(x)P$, the name x is made local to P by the binding prefix '(x)'. The agent $[x{=}y]P$ behaves like P if x and y are the same name and otherwise behaves like $\mathbf{0}$. Finally, the defined agent $A(x_1, ..., x_n)$ behaves like the corresponding instance of the right-hand side of the defining equation for the n-ary agent identifier A.

2.2 The transitional semantics

As in CCS, agents are given a transitional semantics based on labelled transitions of the form $P \xrightarrow{\alpha} Q$, which can be read 'P can perform the action α and then evolve into Q'. In the π-calculus, there are four types of action:

$$\begin{array}{rcll} A & ::= & \tau & \text{silent action} \\ & \mid & \overline{x}y & \text{free output action} \\ & \mid & x(y) & \text{input action} \\ & \mid & \overline{x}(y) & \text{bound output action} \end{array}$$

The silent action arises from internal communication within an agent, as well as from agents of the form $\tau.P$. Output-prefixed agents such as $\overline{x}y.P$ give rise to the free output action $\overline{x}y$, and input prefixed agents $x(y).P$ to the input action $x(y)$. Bound output actions of the form $\overline{x}(y)$ arise from output actions that export a name outside its current scope.

The following notation, which is introduced in [14], is used in defining the transition relation for the π-calculus. The set of bound names of an action α is written $\mathrm{bv}(\alpha)$, and the set of free names of an action α is written $\mathrm{fv}(\alpha)$. The meaning of this notation is defined by:

$$\mathrm{bv}(\alpha) = \begin{cases} \{y\} & \text{if } \alpha = x(y) \text{ or } \overline{x}(y) \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathrm{fv}(\alpha) = \begin{cases} \{x\} & \text{if } \alpha = x(y) \text{ or } \overline{x}(y) \\ \{x, y\} & \text{if } \alpha = \overline{x}y \\ \{\} & \text{if } \alpha = \tau \end{cases}$$

The set of names of an action $\mathrm{v}(\alpha)$ is defined to be the union of $\mathrm{bv}(\alpha)$ and $\mathrm{fv}(\alpha)$. An expression of the form '$P\{y_1, \ldots, y_n/x_1, \ldots, x_n\}$' denotes the result of simultaneously substituting y_i for x_i for $1 \leq i \leq n$ in P, with change of bound names as required to avoid capture.

$$\text{TAU-ACT: } \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \text{OUTPUT-ACT: } \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P}$$

$$\text{INPUT-ACT: } \frac{}{x(z).P \xrightarrow{x(w)} P\{w/z\}} \quad w \notin \mathrm{fv}((z)P)$$

$$\text{SUM: } \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{MATCH: } \frac{P \xrightarrow{\alpha} P'}{[x{=}x]P \xrightarrow{\alpha} P'}$$

$$\text{IDE: } \frac{P\{y_1, \ldots, y_n/x_1, \ldots, x_n\} \xrightarrow{\alpha} P'}{A(y_1, \ldots, y_n) \xrightarrow{\alpha} P'} \quad A(x_1, \ldots, x_n) \stackrel{\mathrm{def}}{=} P$$

$$\text{PAR: } \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \mathrm{bv}(\alpha) \cap \mathrm{fv}(Q) = \{\}$$

$$\text{COM: } \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{y/z\}} \qquad \text{CLOSE: } \frac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P \mid Q \xrightarrow{\tau} (w)(P' \mid Q')}$$

$$\text{RES: } \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'} \quad y \notin \mathrm{v}(\alpha) \qquad \text{OPEN: } \frac{P \xrightarrow{\overline{x}y} P'}{(y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}} \quad \begin{array}{l} y \neq x \\ w \notin \mathrm{fv}((y)P') \end{array}$$

Figure 14: Transition rules for the π-calculus.

The transition relation $P \xrightarrow{\alpha} Q$ itself is defined inductively by the rules shown in figure 14, together with additional symmetric rules for the operators $\mid$ and $+$. More precisely, the three-place relation $\longrightarrow \subseteq (agent \times action \times agent)$ is defined to be the smallest set closed under these rules, where '$(P, \alpha, Q) \in \longrightarrow$' is written '$P \xrightarrow{\alpha} Q$'. The details of the transition rules are not relevant here; they are shown in full merely to give the reader a general idea of the size and complexity of the calculus.

2.3 Bisimulation and Equivalence

As in CCS, equivalence of agents in the π-calculus is defined using the notion of a bisimulation between agents. A binary relation S is a *strong simulation* if $P\ S\ Q$ implies that

1. If $P \xrightarrow{\alpha} P'$ and $\alpha = \tau$ or $\alpha = \overline{x}y$, then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P'\ S\ Q'$.
2. If $P \xrightarrow{x(y)} P'$ and $y \notin \mathrm{v}(P) \cup \mathrm{v}(Q)$, then for some Q', $Q \xrightarrow{x(y)} Q'$ and for all w, $P'\{w/y\}\ S\ Q'\{w/y\}$.
3. If $P \xrightarrow{\overline{x}(y)} P'$ and $y \notin \mathrm{v}(P) \cup \mathrm{v}(Q)$, then for some Q', $Q \xrightarrow{\overline{x}(y)} Q'$ and $P'\ S\ Q'$.

A *strong bisimulation* is a strong simulation S whose inverse is also a strong simulation. The equivalence relation $\dot{\sim}$ is defined to be the largest strong bisimulation, so that two agents P and Q are equivalent if $P\ S\ Q$ for some strong bisimulation S.

The algebraic theory of equivalence for the π-calculus is based on the definitions of strong bisimulation and the equivalence relation $\dot{\sim}$ given above. Many of the algebraic laws for the π-calculus correspond to similar or identical laws in CCS, for example the following equations for summation:

$$
\begin{array}{rcll}
P + \mathbf{0} & \dot{\sim} & P & \qquad\qquad \text{zero} \\
P + P & \dot{\sim} & P & \qquad\qquad \text{idempotence} \\
P_1 + P_2 & \dot{\sim} & P_2 + P_1 & \qquad\qquad \text{commutativity} \\
P_1 + (P_2 + P_3) & \dot{\sim} & (P_1 + P_2) + P_3 & \qquad\qquad \text{associativity}
\end{array}
$$

For a presentation of the full algebraic theory, detailed proofs of soundness and of completeness for finite agents, and for a discussion of other notions of equivalence for the π-calculus, see the reports [13, 14].

3 Mechanizing the π-calculus in HOL

One approach to mechanizing a formal system in HOL is to translate its syntactic objects directly into appropriate denotations in higher order logic. This approach is exemplified by Mike Gordon's work on mechanizing Hoare logic [7]. Meaning is given to partial correctness statements in Hoare logic by translating them into propositions of higher order logic that capture their intended semantics. For example, the partial correctness statement

$$\{\mathtt{X}{=}n\} \quad \mathtt{X:=X+1} \quad \{\mathtt{X}{=}n{+}1\}$$

is translated into the assertion that the following relation holds of any pair of initial and final states s_1 and s_2:

$$\forall n.\, ((s_1\ \mathtt{X} = n) \wedge (s_2 = \lambda v.\, (v{=}\mathtt{X} \Rightarrow (s_1\ \mathtt{X}) + 1 \mid s_1\ v))) \ \supset\ (s_2\ \mathtt{X} = n + 1)$$

Program variables are represented by constants of a specially-defined logical type *var*, and states are modelled by total functions from program variables to natural numbers. Partial correctness statements are represented directly by their denotations in logic—which, with sufficient parser and pretty-printer support, can be made to look like assertions in Hoare logic (see [7]).

The advantage of this approach is that the embedded formal system inherits a certain amount of syntactic infrastructure from the underlying logic. For example, λ-abstraction and β-reduction in higher order logic can be used to simulate variable binding and substitution in the language being mechanized. The result is a system particularly well suited to reasoning about applications, since the HOL system provides highly optimized proof support for these basic syntactic notions.

The disadvantage of direct translation is that it does not allow metatheoretic reasoning about the embedded formal system to be carried out within higher order logic itself. For example, a proposition that makes reference to the embedded language as a whole cannot be expressed in the logic; it can be stated only as a metatheorem about classes of logical assertions and hence cannot be proved in HOL.

One goal of the present work is to support formal metatheoretic reasoning about the π-calculus itself, as well as reasoning in the calculus about applications. A different approach is therefore taken to mechanizing the π-calculus in HOL. The language of agents is embedded as an object (or, more specifically, as a defined type) within the logic, rather than metalinguistically translated into terms of the logic. Higher order logic is thus used as a formal metalanguage whose objects are the process or agent expressions of the π-calculus. Meaning is then given to these expressions by defining the labelled transition semantics, strong bisimulation, and the equivalence relation $\dot{\sim}$ within the logic itself.

A similar approach is taken by Camilleri [2] in his formalization of CSP in higher order logic, by Back and von Wright [1] in their work on mechanized program transformation in HOL, and in the present author's work [11] on reasoning about circuit models. All this work, however, contrives to avoid explicit definitions of substitution, essentially by inheriting it from higher order logic. In this respect, it differs from the present formalization of the π-calculus, in which all syntactic operations over the embedded language of agent expressions are defined within the logic and can therefore be mentioned explicitly in propositions of the logical metalanguage.

4 Embedding the syntax of agents in HOL

This section outlines the development of a definitional HOL theory of the language of agent expressions in the π-calculus. For clarity of notation, as well as for fidelity to the presentation in [13, 14], the theory makes use of a predefined logical type $(\alpha)set$, values of which are sets of elements of type α. This type is defined formally in the built-in HOL 'set theory' library, which contains a substantial collection of basic theorems about sets. Also provided by the library are parser and pretty-printer support for naming finite sets by enumeration, for example by writing '$\{a, b, c\}$', and for the set specification notation '$\{x \mid \varphi(x)\}$'. Sets written in these notations should be regarded as metalinguistic abbreviations, expanded by the HOL term parser into corresponding logical terms.

4.1 Representing names in logic

An obvious way to represent names in higher order logic is to model the set of names $\mathcal{N}$ by a logical type. The only property required of $\mathcal{N}$ is that it must be infinite, so that bound names can always be changed to avoid capture of names by the binding constructs '$x(y).P$' and '$(y)P$' when a substitution is done. Names can therefore be represented in logic by any type that contains an infinite number of distinct values—for example the type of natural numbers.

But rather than develop the theory with a particular fixed representation for names, the set of names $\mathcal{N}$ is represented by a type variable 'α'. An infinite set of names is then assumed by working (when necessary) under the hypothesis that there exists a choice function $ch{:}(\alpha)set{\to}\alpha$ which for any finite set of names S yields a name not in S:

$$\forall S.\ \mathsf{Finite}\ S \supset \neg(ch\ S \in S)$$

This *infinity hypothesis* is required only for the proofs of certain theorems about the π-calculus whose truth depends on the ability to change bound names during substitution. The assumption that there exists a choice function ch with the above property is provably equivalent in HOL to the alternative hypothesis:

$$\exists f{:}\alpha{\to}\alpha.\ (\forall x\ y.\ (f\ x = f\ y) \supset (x = y)) \wedge (\forall y.\ \exists x.\ \neg(f\ x = y))$$

This asserts the type α has no more elements than some proper subset of α—i.e. it asserts of α that it satisfies the conventional definition of an infinite set.

The use of a type variable to represent the set of names $\mathcal{N}$ results in a polymorphic theory of the π-calculus in HOL. The entire theory can therefore be specialized for a particular application by choosing an (infinite) application-specific logical type to model names, instantiating the type variable α to this type, and discharging the resulting infinity hypothesis wherever it appears. This is not an atomic operation in the HOL system, but it is not hard to program in ML. The only part that cannot be automated is proving the existence of a choice function for the type selected to represent names.

4.2 Defining the syntax of agents

The formal language of agents in the π-calculus is embedded in HOL by defining a logical type $(\alpha)agent$, values of which represent agent expressions with names of type α. The primitive rule of type definition, as was mentioned in section 1.1.1, allows new types to be introduced into the logic only as names for subsets of already existing types. So to define a type of agent expressions a rather complex encoding into values of an existing logical type is required.

The HOL system, however, provides a derived principle of definition that automates all the formal inference necessary to define an arbitrary concrete recursive type in higher order logic [10]. The user supplies a specification of the required type in a form similar to a datatype declaration in Standard ML [9]. The system then constructs an appropriate encoding for values of the required type, defines the type using this encoding and the primitive rule of type definition, and automatically proves an abstract characterization of the newly-defined type. The details of the definition are hidden from the user, and the ML implementation is highly optimized; to the user, this derived principle of *recursive type definition* appears almost primitive.

Using the derived rule of recursive type definition, the language of agent expressions is embedded in logic by the type $(\alpha)agent$ specified by:

$agent$	$::=$	Zero	Zero represents $\mathbf{0}$
	$\mid$	$\mathsf{Neg}\ \alpha\ \alpha\ agent$	$\mathsf{Neg}\ x\ y\ P$ represents $\overline{x}y.P$
	$\mid$	$\mathsf{Pos}\ \alpha\ \alpha\ agent$	$\mathsf{Pos}\ x\ y\ P$ represents $x(y).P$
	$\mid$	$\mathsf{Tau}\ agent$	$\mathsf{Tau}\ P$ represents $\tau.P$
	$\mid$	$\mathsf{Plus}\ agent\ agent$	$\mathsf{Plus}\ P_1\ P_2$ represents $P_1 + P_2$
	$\mid$	$\mathsf{Comp}\ agent\ agent$	$\mathsf{Comp}\ P_1\ P_2$ represents $P_1 \mid P_2$
	$\mid$	$\mathsf{Res}\ \alpha\ agent$	$\mathsf{Res}\ x\ P$ represents $(x)P$
	$\mid$	$\mathsf{Match}\ \alpha\ \alpha\ agent$	$\mathsf{Match}\ x\ y\ P$ represents $[x{=}y]P$
	$\mid$	$\mathsf{Repl}\ agent$	$\mathsf{Repl}\ P$ represents $!P$

This equation specifies a concrete recursive type with nine constructors, each of which (except Repl, which is explained later) corresponds to one of the forms of agent expression in the π-calculus syntax presented in section 2.1. Given this specification, the rule of recursive type definition automatically finds a representation for the required type $(\alpha)agent$ and makes an appropriate primitive type definition for it. The system also makes an appropriate constant definition for each of the specified constructors. Plus, for example, becomes a constant of type

$$(\alpha)agent \to (\alpha)agent \to (\alpha)agent$$

introduced by means of the primitive rule of constant definition. Likewise, Res becomes a constant function that maps a value of type α representing a name to a value of type $(\alpha)agent$ representing an agent—and so on.

The result is a single theorem of higher order logic which provides a complete and abstract characterization of the type $(\alpha)agent$ and forms the basis for all further reasoning about it. The theorem asserts the admissibility of defining functions over agents by primitive recursion:

$$
\begin{array}{l}
\vdash \forall e\ f_0\ f_1\ f_2\ f_3\ f_4\ f_5\ f_6\ f_7. \\
\quad \exists! fn{:}(\alpha)agent{\rightarrow}\beta. \\
\qquad fn\ \mathsf{Zero} = e\ \wedge \\
\qquad \forall x_0\ x_1\ a.\ fn(\mathsf{Neg}\ x_0\ x_1\ a) = f_0\ (fn\ a)\ x_0\ x_1\ a\ \wedge \\
\qquad \forall x_0\ x_1\ a.\ fn(\mathsf{Pos}\ x_0\ x_1\ a) = f_1\ (fn\ a)\ x_0\ x_1\ a\ \wedge \\
\qquad \forall a.\ fn(\mathsf{Tau}\ a) = f_2\ (fn\ a)\ a\ \wedge \\
\qquad \forall a_1\ a_2.\ fn(\mathsf{Plus}\ a_1\ a_2) = f_3\ (fn\ a_1)\ (fn\ a_2)\ a_1\ a_2\ \wedge \\
\qquad \forall a_1\ a_2.\ fn(\mathsf{Comp}\ a_1\ a_2) = f_4\ (fn\ a_1)\ (fn\ a_2)\ a_1\ a_2\ \wedge \\
\qquad \forall x\ a.\ fn(\mathsf{Res}\ x\ a) = f_5\ (fn\ a)\ x\ a\ \wedge \\
\qquad \forall x_0\ x_1\ a.\ fn(\mathsf{Match}\ x_0\ x_1\ a) = f_6\ (fn\ a)\ x_0\ x_1\ a\ \wedge \\
\qquad \forall a.\ fn(\mathsf{Repl}\ a) = f_7\ (fn\ a)\ a
\end{array}
$$

This is an abstract characterization of the language of agents in logic which is both succinct and complete, in the sense that it completely determines the structure of agent expressions up to isomorphism. It can be viewed as slight extension of the *initiality* property by which structures are characterized in the 'initial algebra' approach to specifying abstract data types [5].

4.2.1 Primitive recursion over agents

As was discussed in section 1.1.1, function constants that satisfy recursive defining equations are not directly definable by the primitive rule for constant definitions. To define such a constant, one must first prove that there in fact exists a total function that satisfies the required recursive equation. The HOL system, however, has a built-in derived principle of *primitive recursive function definition*, which automates existence proofs for primitive recursive functions defined over concrete recursive types such as $(\alpha)agent$.

Given the characterizing theorem for $(\alpha)agent$ and the primitive recursive defining equations for a function over agents, this rule automatically proves the existence of a total function that satisfies these equations. A constant is then introduced by a constant specification to name this total function. The details of the proofs are hidden from the user, who for all practical purposes can simply regard this derived principle of recursive function definition as part of the primitive basis of the logic.

4.2.2 Structural induction on agents

The HOL system also has a built-in derived inference rule for proving a structural induction theorem for any concrete recursive type. Given the recursion theorem for $(\alpha)agent$ shown above, this rule automatically proves a theorem which states the validity of structural induction on agent expressions. This induction theorem can, in turn, be used with another built-in proof tool to automatically construct a HOL *tactic* for interactive goal-directed proofs by structural induction on agents. (See any one of [4, 6, 8, 15] for an explanation of tactics.) As one might expect, this tactic is invaluable for proving many of the basic syntactic theorems about the π-calculus in HOL.

4.3 Agent identifiers and replication

The π-calculus syntax shown in section 2.1 includes defined agents of the form $A(x_1, \ldots, x_n)$, where A is an n-ary agent identifier. Agent identifiers, together with their defining equations, supply the π-calculus both with object-language abbreviations for agent expressions and with recursion. The latter is the essential function of agent identifiers; without them, there is no way to express infinite behaviour. In the HOL mechanization, however, agent identifiers are replaced by an alternative way of providing unbounded behaviour, namely the *replication* of agents. This difference represents the only significant point (at least, so far) at which the principle that the HOL theory should be as close as possible to the calculus as presented in [13, 14] has been compromised.

The replication of an agent P is written '$!P$' and is represented in logic by '$\mathsf{Repl}\ P$'. The agent $!P$ can be thought of as the parallel composition of as many instances of P as desired. This is reflected in the following transition rule for replication

$$\text{REPL:} \quad \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

which states that whatever action can be performed by the parallel composition of an agent P with the replication $!P$ can also be done by the replication $!P$ itself. In the HOL theory of the π-calculus, this rule replaces the agent identifier rule IDE shown in figure 14.

Replacing agent identifiers by replication considerably simplifies the HOL mechanization. It avoids the need to parameterize the entire theory by sets of defining equations and to work under well-formedness hypotheses for these equations. But for many applications, recursive agent definitions are likely to be more direct and natural to use than replication. The theory aims, therefore, to recover at least some of the utility of agent identifiers. The merely abbreviatory role of agent identifiers can just be transferred to ordinary constant definitions. But the expressive power of recursive defining equations can be regained only at the cost of developing some special-purpose proof support. The aim is eventually to support recursive agent definitions by a method similar to that by which recursive function definitions are automated in HOL. Preliminary experiments indicate that this approach is feasible, but little work has been done in this area so far.

4.4 Elementary syntactic theory

Having defined the type $(\alpha)agent$ in logic, it is straightforward, if rather tedious, to develop the elementary theory of the syntax of agents in HOL. This comprises the various definitions and theorems about free and bound names, substitution, and α-equivalence of agents needed for later proofs—matters that are covered in a mere page or so in the report [14], but which naturally takes considerably longer to treat formally. The following sections outline the HOL theory of free and bound names and substitution; the definition of α-equivalence is omitted.

4.4.1 Free and bound names

Development of the theory begins with defining the function constants Fv, Bv and V. These have the logical type $(\alpha)agent{\rightarrow}(\alpha)set$ and correspond to the functions fv, bv and v described above in section 2.1. The definitions use some of the infrastructure provided by the HOL set theory library, namely the basic operations of set union and set difference, as well as notation for specifying finite sets by enumeration. The functions themselves are primitive recursive over the type of agent expressions $(\alpha)agent$. They can therefore be defined simply by supplying the required defining equations to the HOL derived rule of recursive function definition. The recursive definition of $\mathsf{Fv}(P)$, for example, is given by the theorem

$$
\begin{aligned}
\vdash\ & \mathsf{Fv}\ \mathsf{Zero} = \{\} \ \wedge \\
& \forall x\ y\ P.\ \mathsf{Fv}(\mathsf{Neg}\ x\ y\ P) = \{x,y\} \cup (\mathsf{Fv}\ P) \ \wedge \\
& \forall x\ y\ P.\ \mathsf{Fv}(\mathsf{Pos}\ x\ y\ P) = \{x\} \cup ((\mathsf{Fv}\ P) - \{y\}) \ \wedge \\
& \forall P.\ \mathsf{Fv}(\mathsf{Tau}\ P) = \mathsf{Fv}\ P \ \wedge \\
& \forall P\ Q.\ \mathsf{Fv}(\mathsf{Plus}\ P\ Q) = (\mathsf{Fv}\ P) \cup (\mathsf{Fv}\ Q) \ \wedge \\
& \forall P\ Q.\ \mathsf{Fv}(\mathsf{Comp}\ P\ Q) = (\mathsf{Fv}\ P) \cup (\mathsf{Fv}\ Q) \ \wedge \\
& \forall x\ P.\ \mathsf{Fv}(\mathsf{Res}\ x\ P) = (\mathsf{Fv}\ P) - \{x\} \ \wedge \\
& \forall x\ y\ P.\ \mathsf{Fv}(\mathsf{Match}\ x\ y\ P) = \{x,y\} \cup (\mathsf{Fv}\ P) \ \wedge \\
& \forall P.\ \mathsf{Fv}(\mathsf{Repl}\ P) = \mathsf{Fv}\ P
\end{aligned}
$$

which is proved automatically by this derived rule, as outlined above in section 4.2.1. The definitions of Bv and V are similar.

A collection of theorems about free and bound names in the π-calculus has been proved in HOL from the definitions of Fv, Bv and V . These theorems are mostly very simple and their proofs trivial; two illustrative examples are:

$$
\begin{aligned}
& \vdash \forall P.\ \mathsf{Finite}(\mathsf{Fv}\ P) \\
& \vdash \forall P.\ \mathsf{V}\ P = \mathsf{Fv}\ P \cup \mathsf{Bv}\ P
\end{aligned}
$$

Both theorems are proved by structural induction on the agent P using the tactic discussed above in section 4.2.2. The significance of first theorem has to do with the need to change bound names to avoid capture during substitution. A fresh name is sometimes needed, distinct from all the names free in a given agent P, and this is possible only if the set $\mathsf{Fv}(P)$ is finite. The second theorem merely states that the function V, which is defined recursively in HOL, satisfies the more direct definition used in [14].

4.4.2 Substitution

One of the more complex definitions in the syntactic theory is the definition of simultaneous substitution of names for free occurrences of names in an agent. The complexity is due, of course, to the name binding constructs of the π-calculus. Bound names sometimes have to be changed to avoid the capture of names introduced by substitution. Furthermore, the present theory of substitution is designed with future use for applications in mind, so bound names are changed only when strictly necessary. This further complicates the definition.

To formalize substitution for the π-calculus in logic, a function

$$\mathsf{Sub} : \underbrace{(\alpha{\to}(\alpha)set{\to}\alpha)}_{\text{choice function}} \to \underbrace{(\alpha)agent}_{\text{agent}} \to \underbrace{(\alpha{\to}\alpha)}_{\text{name mapping}} \to \underbrace{(\alpha)agent}_{\text{result}}$$

is defined by primitive recursion on agents. The function Sub takes two arguments in addition to the agent in which the substitution is to be done. One is a name mapping $s{:}\alpha{\to}\alpha$, which specifies the particular substitition of names for names required. The other argument is a choice function $ch{:}\alpha{\to}(\alpha)set{\to}\alpha$, which is used in the body of the definition of substitution to generate fresh names wherever a change of bound names is required. It is assumed that the choice function has the property that for any name n and finite set of names S, the name $ch\ n\ S$ is not an element of S. This is expressed by the proposition

$$\forall S.\ \mathsf{Finite}\ S \supset \forall n.\ \neg(ch\ n\ S \in S)$$

which is taken as a hypothesis, if necessary, in proofs involving substitution—as was discussed above in section 4.1. In general, the choice function is assumed to take both a name n and a set S as arguments. This is done so that application-specific instances of the choice function can, if desired, generate a name not in S by taking some variant of the name n.

The primitive recursive definition of Sub in HOL is given by the theorem shown below. The notation '$\texttt{let}\ v = t_1\ \texttt{in}\ t_2$' used in this definition is a metalinguistic abbreviation supported by the HOL parser and pretty-printer. It expands into a term provably equivalent to $(\lambda v.\, t_2)\ t_1$.

$$
\begin{array}{l}
\vdash \forall ch\ s.\ \mathsf{Sub}\ ch\ \mathsf{Zero}\ s = \mathsf{Zero} \wedge \\
\quad \forall ch\ x\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Neg}\ x\ y\ P)\ s = \mathsf{Neg}\ (s\ x)\ (s\ y)\ (\mathsf{Sub}\ ch\ P\ s) \wedge \\
\quad \forall ch\ x\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Pos}\ x\ y\ P)\ s\ = \\
\qquad\qquad \texttt{let}\ vs = \mathsf{Image}\ s\ ((\mathsf{Fv}\ P) - \{y\})\ \texttt{in} \\
\qquad\qquad \texttt{let}\ y' = (y \in vs \Rightarrow ch\ y\ vs \mid y)\ \texttt{in} \\
\qquad\qquad\quad \mathsf{Pos}\ (s\ x)\ y'\ (\mathsf{Sub}\ ch\ P\ (\lambda n.\ (n{=}y) \Rightarrow y' \mid s\ n)) \wedge \\
\quad \forall ch\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Tau}\ P)\ s = \mathsf{Tau}\ (\mathsf{Sub}\ ch\ P\ s) \wedge \\
\quad \forall ch\ P\ Q\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Plus}\ P\ Q)\ s = \mathsf{Plus}\ (\mathsf{Sub}\ ch\ P\ s)\ (\mathsf{Sub}\ ch\ Q\ s) \wedge \\
\quad \forall ch\ P\ Q\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Comp}\ P\ Q)\ s = \mathsf{Comp}\ (\mathsf{Sub}\ ch\ P\ s)\ (\mathsf{Sub}\ ch\ Q\ s) \wedge \\
\quad \forall ch\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Res}\ y\ P)\ s = \\
\qquad\qquad \texttt{let}\ vs = \mathsf{Image}\ s\ ((\mathsf{Fv}\ P) - \{y\})\ \texttt{in} \\
\qquad\qquad \texttt{let}\ y' = (y \in vs \Rightarrow ch\ y\ vs \mid y)\ \texttt{in} \\
\qquad\qquad\quad \mathsf{Res}\ y'\ (\mathsf{Sub}\ ch\ P\ (\lambda n.\ (n = y) \Rightarrow y' \mid s\ n)) \wedge \\
\quad \forall ch\ x\ y\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Match}\ x\ y\ P)\ s = \mathsf{Match}\ (s\ x)\ (s\ y)\ (\mathsf{Sub}\ ch\ P\ s) \wedge \\
\quad \forall ch\ P\ s.\ \mathsf{Sub}\ ch\ (\mathsf{Repl}\ P)\ s = \mathsf{Repl}\ (\mathsf{Sub}\ ch\ P\ s)
\end{array}
$$

The definition is straightforward, except for the defining equations for the input prefix Pos and restriction Res. For all the other constructors, the function Sub simply maps the substitution recursively down through an agent, applying the mapping s wherever free names occur. The input prefix and restriction constructs '$\mathsf{Pos}\ x\ y\ P$' and '$\mathsf{Res}\ y\ P$', however, both bind the name y. It may therefore be necessary to change this bound name to a fresh name y', in order to avoid capture of names when the substitution s is applied to P. The definition ensures that bound names are changed only when necessary, namely when y occurs in the image of the function s on the set of all names (other than y itself) that occur free in P. In this case, the bound name is changed to a new name y' which is generated by the choice function ch and which, under the infinity hypothesis for ch, does not occur in this set. Any free occurrences of y in P are also changed to y'.

4.4.3 Theorems about substitution

A number of general theorems about substitution are needed for proofs about the π-calculus. The content of these theorems is mostly predictable, and a full list of theorems need not be given here. In proving these theorems in the HOL system, care was taken restrict dependence on the infinity hypothesis for the choice function to only those theorems for which it is really needed. For example, one of the theorems proved in HOL states that the identity substitution leaves agents unchanged:

$$\vdash \forall P\ ch.\ \mathsf{Sub}\ ch\ P\ (\lambda x.x) = P$$

This proposition holds for any function ch whatsoever, and the theorem therefore does not include the infinity hypothesis for ch as an assumption. By contrast, the following theorem

$$\vdash \forall ch.\ (\forall S.\ \mathsf{Finite}\ S \supset \forall n.\ \neg(ch\ n\ S \in S))\ \supset \forall P\ s.\ \mathsf{Fv}\ (\mathsf{Sub}\ ch\ P\ s) = \mathsf{Image}\ s\ (\mathsf{Fv}\ P)$$

states that the set of names that occur free in an agent after substitution with a name mapping s is the same as the image of the function s on the original set of free names. This holds only if the choice function ch correctly generates new bound names and, of course, if the set of names α is infinite. In this theorem, the infinity hypothesis is essential.

4.4.4 Substitution for a single name

Simultaneous substitution of names for names is needed for only certain parts of the theory of the π-calculus developed in [13, 14]. In the absence of agent identifiers, full simultaneous substitution is not needed for defining the transition relation, strong bisimulation and the equivalence relation $\dot{\sim}$. Substitution for a single name will suffice.

Substitution of x for y in the agent P, written '$P\{x/y\}$' in the notation of section 2.1, is formalized by the constant definition

$$\vdash \forall ch\ P\ x\ y.\ \mathsf{Sub1}\ ch\ P\ (x,y) = \mathsf{Sub}\ ch\ P\ (\lambda n.\ (n = y) \Rightarrow x \mid n)$$

where substitution for a single name is defined in terms of a simultaneous substitution in which the name mapping is the identity function on all names but one. Theorems about the special case of substitution for a single name are (mostly) straightforward to prove in HOL, given this definition of Sub1 and the more general theory of simultaneous substitution.

5 Formalizing the transitional semantics

The theory outlined above provides all the syntactic infrastructure needed to define and reason about the transitional semantics for the π-calculus in logic. This section describes how the labelled transition relation on which this semantics is based is defined in HOL and gives a sketch of the theory developed from this definition.

5.1 Representing actions in HOL

The transition system for the π-calculus shown in section 2.2 is based on four kinds of actions. These are represented in logic by values of the type $(\alpha)action$, which is specified by:

$$\begin{array}{rcll} action & ::= & \mathsf{tau} & \mathsf{tau}\text{ represents } \tau \\ & \mid & \mathsf{fo}\ \alpha\ \alpha & \mathsf{fo}\ x\ y \text{ represents } \overline{x}y \\ & \mid & \mathsf{in}\ \alpha\ \alpha & \mathsf{in}\ x\ y \text{ represents } x(y) \\ & \mid & \mathsf{bo}\ \alpha\ \alpha & \mathsf{bo}\ x\ y \text{ represents } \overline{x}(y) \end{array}$$

and which is defined automatically using the same derived rule of (recursive) type definition used to define the type of agents. The concrete type $(\alpha)action$ specified by this equation has four constructors. One of these (namely tau) is a constant representing the distinguished action τ; the other three are functions of type $\alpha{\rightarrow}\alpha{\rightarrow}(\alpha)action$ that map a pair of names to the representation of an action.

Given this specifying equation for the type of actions, the derived rule of type definition automatically proves the following characterizing theorem for the type $(\alpha)action$:

$$\begin{array}{l}\vdash \forall e\ f_0\ f_1\ f_2.\ \exists! fn{:}(\alpha)action{\rightarrow}\beta. \\ \quad fn\ \mathsf{tau} = e\ \wedge \\ \quad \forall x_0\ x_1.\ fn(\mathsf{fo}\ x_0\ x_1) = f_0\ x_0\ x_1\ \wedge \\ \quad \forall x_0\ x_1.\ fn(\mathsf{in}\ x_0\ x_1) = f_1\ x_0\ x_1\ \wedge \\ \quad \forall x_0\ x_1.\ fn(\mathsf{bo}\ x_0\ x_1) = f_2\ x_0\ x_1\end{array}$$

This theorem asserts that functions over the type $(\alpha)action$ can be uniquely defined by cases on the four different kinds of actions in the π-calculus. It is straightforward to use this theorem in conjunction with the derived principle of (primitive recursive) function definition to define logical counterparts to the functions fv, bv and v on actions introduced in section 2.2. For example, the definition of a function $\mathsf{fv}{:}(\alpha)action{\rightarrow}(\alpha)set$ that corresponds to fv is just:

$$\begin{array}{l}\vdash \mathsf{fv}\ \mathsf{tau} = \{\}\ \wedge \\ \quad \forall x\ y.\ \mathsf{fv}(\mathsf{fo}\ x\ y) = \{x,y\}\ \wedge \\ \quad \forall x\ y.\ \mathsf{fv}(\mathsf{in}\ x\ y) = \{x\}\ \wedge \\ \quad \forall x\ y.\ \mathsf{fv}(\mathsf{bo}\ x\ y) = \{x\}\end{array}$$

The definitions of functions bv and v corresponding to bv and v are similar. Given these definitions and the characterizing theorem for the type $(\alpha)action$, it is trivial to develop a basic theory of actions for the π-calculus in HOL.

5.2 Defining the labelled transition relation

In [14], the transition relation $\longrightarrow$ is defined inductively by the rules reproduced in the present paper in figure 14. In the mechanized theory of the π-calculus this relation is also defined inductively, using a derived principle of *inductive predicate definition* recently implemented (by the present author) in HOL. Given the user's specification of a desired set of rules, this derived principle of definition automatically proves the existence of the relation inductively defined by them. More precisely, the system constructs a term that explicitly denotes the smallest relation closed under the rules specified by the user. HOL then introduces (via a constant specification) a constant to name this relation. The result is a collection of automatically proved theorems stating that the newly-defined relation is in fact closed under the required rules, together with an additional theorem asserting that it is the smallest such relation.

To define the transition relation using this derived principle of inductive definition, the user just enters the transition rules shown in figure 14 as a list of pairs of the form:

$$(\,\langle \textit{list of premises}\rangle, \langle \textit{conclusion}\rangle\,)$$

Each pair consists of a list of the premises of a rule, including any side conditions, and its conclusion. There is one such pair for each of the transition rules, including all symmetric forms. The premises and conclusions are stated using the HOL representation of agents and actions and (where necessary) the notation for free and bound names and substitution defined in the syntactic theory described above.

Given this user-supplied specification of the rules, the system constructs a logical statement of each transition rule in the form of an implication of conclusion by premises. These express what it means for a relation $R{:}(\alpha)agent{\rightarrow}(\alpha)action{\rightarrow}(\alpha)agent{\rightarrow}bool$ to be closed under each of the rules. The assertion that the relation R is closed under the left-hand symmetric form of the SUM rule, for example, is expressed in logic by the implication shown below.

$$\forall P\ a\ P'.\ R\ P\ a\ P' \supset \forall Q.\ R\ (\mathsf{Plus}\ P\ Q)\ a\ P'$$

Likewise, the translation into higher order logic of the OPEN rule is:

$$\begin{array}{l}\forall P\ x\ y\ P'\ w.\\ \quad R\ P\ (\mathsf{fo}\ x\ y)\ P' \wedge \neg(y{=}x) \wedge \neg w \in \mathsf{Fv}(\mathsf{Res}\ y\ P')\ \supset\\ \quad\quad R\ (\mathsf{Res}\ y\ P)\ (\mathsf{bo}\ x\ w)\ (\mathsf{Sub1}\ ch\ P'\ (w,y))\end{array}$$

This logical formulation of the OPEN rule illustrates an explicit use of the defined syntactic notions of substitution and the set of free names in an agent. The translations into logic of the remaining rules are similar to these examples.

The definition made by HOL of the transition relation is based on this translation of the rules into logical implications. The conjunction of all these implications asserts the closure of an arbitrary three-place relation R under the transition rules of the π-calculus, and the labelled transition relation itself is just defined to be the intersection of all such relations. More precisely, the derived HOL rule of inductive definition makes a constant specification for the relation

$$\mathsf{Trans} : (\alpha{\rightarrow}(\alpha)set{\rightarrow}\alpha) \rightarrow (\alpha)agent \rightarrow (\alpha)action \rightarrow (\alpha)agent \rightarrow bool$$

which is logically equivalent to the following constant definition:

$$\begin{array}{l}\vdash \mathsf{Trans}\ ch\ P\ a\ Q =\\ \quad \forall R{:}(\alpha)agent{\rightarrow}(\alpha)action{\rightarrow}(\alpha)agent{\rightarrow}bool.\\ \quad\quad \langle R\ \textit{is closed under the rules}\rangle \supset R\ P\ a\ Q\end{array}$$

This definition states that there is a transition from the agent P to the agent Q labelled by the action a exactly when P, a and Q are in the intersection (i.e. '$\forall$') of every relation R closed under the transition rules for the π-calculus. The relation Trans must take the choice function ch as an argument, since substitution is employed in stating closure under the rules.

The final result of making the automatic inductive definition sketched above (and all the user actually sees) is a collection of theorems that state the transition rules for the defined relation Trans, together with a theorem stating that Trans is the smallest relation closed under these rules. The following theorems for the left-hand SUM rule and the OPEN rule, for example, are among the theorems proved automatically by the system:

$$\vdash \forall ch\ P\ a\ P'.\ \mathsf{Trans}\ ch\ \ P\ a\ P' \supset \forall Q.\ \mathsf{Trans}\ ch\ (\mathsf{Plus}\ P\ Q)\ a\ P'$$

$$\begin{array}{l}\vdash \forall ch\ P\ x\ y\ P'\ w.\\ \quad \mathsf{Trans}\ ch\ P\ (\mathsf{fo}\ x\ y)\ P' \wedge \neg(y{=}x) \wedge \neg w \in \mathsf{Fv}(\mathsf{Res}\ y\ P')\ \supset\\ \quad\quad \mathsf{Trans}\ ch\ (\mathsf{Res}\ y\ P)\ (\mathsf{bo}\ x\ w)\ (\mathsf{Sub1}\ ch\ P'\ (w,y))\end{array}$$

There are sixteen such theorems for the π-calculus with replication in place of agent identifiers, one for each transition rule including symmetric forms. The additional theorem stating that

Trans (actually, that $\mathsf{Trans}\ ch$) is the smallest relation closed under the rules, which is also derived automatically by the rule of inductive predicate definition, has the form:

$$\begin{array}{l} \vdash \forall ch.\ \forall R{:}(\alpha)agent{\rightarrow}(\alpha)action{\rightarrow}(\alpha)agent{\rightarrow}bool. \\ \qquad \langle R \textit{ is closed under the rules}\rangle \supset \\ \qquad\quad \forall P\ a\ Q.\ \mathsf{Trans}\ ch\ P\ a\ Q \supset R\ P\ a\ Q \end{array}$$

This *rule induction theorem* for Trans is essential for proving properties of the transition relation by induction on the depth of inference (i.e. by *rule induction*). By appeal to an appropriate instance of this theorem, one may reduce proving that some property $R[P,a,Q]$ holds of all a-labelled transitions from P to Q to showing that this property is preserved by the transition rules for the π-calculus.

5.3 Proof tools associated with the transition relation

Associated with the derived rule of inductive predicate definition are several general-purpose proof tools for reasoning about inductively defined relations in HOL. The most important of these is a tactic for interactive goal-directed proofs by rule induction. This tactic mechanizes the inductive form of argument outlined above; given the rule induction theorem for Trans and a hypothesis to be proved of the form:

$$\forall P\ a\ Q.\ \mathsf{Trans}\ ch\ P\ a\ Q \supset R[P,a,Q]$$

the rule induction tactic reduces the task of proving this hypothesis to proving that the property expressed by '$R[P,a,Q]$' is preserved by the rules that inductively define Trans. Many of the proofs about the π-calculus in the report [14] are done by induction on the depth of inference, so this tactic is of primary importance in mechanizing these proofs in HOL.

Other proof tools associated with the transition relation include a set of HOL tactics for proving that specific labelled transitions hold between agents of the calculus. For example, one of these tactics can be used to reduce the task of proving that $\mathsf{Trans}\ ch\ (P+Q)\ a\ P'$ to proving that $\mathsf{Trans}\ ch\ P\ a\ P'$. These tactics are constructed automatically by the system from the theorems stating the transition rules for Trans. There is also an automatic proof procedure for deriving an exhaustive case analysis theorem for the transition system:

$\vdash \mathsf{Trans}\ ch\ P\ a\ Q\ \supset$	if $P \xrightarrow{a} Q$, then this follows from
$\quad (P{=}\mathsf{Tau}\ Q \wedge a{=}\mathsf{tau}) \vee$	the TAU-ACT rule, or
$\quad (\exists x\ y.\ P{=}\mathsf{Neg}\ x\ y\ Q \wedge a{=}\mathsf{fo}\ x\ y) \vee$	the OUTPUT-ACT rule, or
$\quad (\exists P'\ Q'.\ P{=}\mathsf{Plus}\ P'\ Q' \wedge \mathsf{Trans}\ ch\ P'\ a\ Q) \vee \ldots$	the PLUS rule, or...

This is used to mechanize arguments about the transition system of the kind that are typically accompanied by an explanation of the form 'if ..., then by a shorter inference ...'.

5.4 Theorems about the transition relation

The theorems and proof tools described above provide the necessary logical infrastructure to develop the HOL theory of the labelled transition relation for the π-calculus. Work on this theory is in its early stages, and the theory is still far from complete. One example of the theorems proved to date is the following lemma about free and bound names, which shows how dependence on the infinity hypothesis propagates to the level of transitions

$$\begin{array}{l} \vdash \forall ch.\ (\forall S.\ \mathsf{Finite}\ s \supset \forall n.\ \neg(ch\ n\ S) \in S)\ \supset \\ \qquad \forall P\ a\ P'.\ \mathsf{Trans}\ ch\ P\ a\ P' \supset (\mathsf{Fv}\ P' \subseteq (\mathsf{Fv}\ P \cup \mathsf{bv}\ a)) \wedge (\mathsf{fv}\ a \subseteq \mathsf{Fv}\ P) \end{array}$$

This is one in a series of lemmas for the proof that α-equivalence is a strong bisimulation presented in the π-calculus report [14]. The HOL proof was done using the rule induction tactic described above. It closely follows the detailed proof given in [14], which proceeds by induction on the depth of inference.

Other theorems that have been proved in HOL about the labelled transition system include various equivalences between transitions, for example:

$$\vdash \forall ch\ P\ a\ Q\ R.\ \mathsf{Trans}\ ch\ (\mathsf{Plus}\ P\ Q)\ a\ R = \mathsf{Trans}\ ch\ (\mathsf{Plus}\ Q\ P)\ a\ R$$

$$\vdash \forall ch\ P\ a\ Q\ R.\ \mathsf{Trans}\ ch\ (\mathsf{Plus}\ P\ Q)\ a\ R = \mathsf{Trans}\ ch\ P\ a\ R \vee \mathsf{Trans}\ ch\ Q\ a\ R$$

$$\vdash \forall ch\ P\ x\ a\ Q.\ \mathsf{Trans}\ ch\ (\mathsf{Match}\ x\ x\ P)\ a\ Q = \mathsf{Trans}\ ch\ P\ a\ Q$$

Simple theorems of this kind follow directly from the rules defining the relation Trans and the case analysis theorem discussed in the preceding section. They are easy to prove, and the proofs are very regular and could be completely automated in HOL.

6 Defining equivalence

Once the substitution function Sub1 and the transition relation Trans have been defined, it is straightforward to express the concept of a strong simulation in logic. The following definition is a direct translation into higher order logic of the definition of strong simulation given in section 2.3.

$$\begin{array}{l} \vdash \mathsf{Sim}\ ch\ S = \\ \quad \forall P\ Q.\ S\ P\ Q\ \supset \\ \qquad \forall P'.\ \mathsf{Trans}\ ch\ P\ \mathsf{tau}\ P'\ \supset \\ \qquad\quad \exists Q'.\ \mathsf{Trans}\ v\ Q\ (\mathsf{fo}\ x\ y)\ Q' \wedge S\ P'\ Q' \wedge \\ \qquad \forall x\ y\ P'.\ \mathsf{Trans}\ ch\ P\ (\mathsf{fo}\ x\ y)\ P'\ \supset \\ \qquad\quad \exists Q'.\ \mathsf{Trans}\ v\ Q\ (\mathsf{fo}\ x\ y)\ Q' \wedge S\ P'\ Q' \wedge \\ \qquad \forall x\ y\ P'.\ \mathsf{Trans}\ ch\ P\ (\mathsf{in}\ x\ y)\ P' \wedge \neg(y \in (\mathsf{V}\ P \cup \mathsf{V}\ Q))\ \supset \\ \qquad\quad \exists Q'.\ \mathsf{Trans}\ ch\ Q\ (\mathsf{in}\ x\ y)\ Q' \wedge \forall w.\ S\ (\mathsf{Sub1}\ ch\ P'\ (w,y))\ (\mathsf{Sub1}\ ch\ Q'\ (w,y)) \wedge \\ \qquad \forall x\ y\ P'.\ \mathsf{Trans}\ ch\ P\ (\mathsf{bo}\ x\ y)\ P' \wedge \neg(y \in (\mathsf{V}\ P \cup \mathsf{V}\ Q))\ \supset \\ \qquad\quad \exists Q'.\ \mathsf{Trans}\ ch\ Q\ (\mathsf{bo}\ x\ y)\ Q' \wedge S\ P'\ Q' \end{array}$$

This defines 'Sim ch S' to mean 'the relation S is a strong simulation'. The predicate Sim must take the choice function ch as a parameter, since its definition depends on substitution.

Given this definition, the equivalence relation $\dot{\sim}$ between agents is defined in HOL by

$$\vdash \mathsf{Equiv}\ ch\ P\ Q = \exists S.\ S\ P\ Q \wedge \mathsf{Sim}\ ch\ S \wedge \mathsf{Sim}\ ch\ (\lambda x\ y.\ S\ y\ x)$$

This definition states that two agents P and Q are equivalent if $S\ P\ Q$ holds for any strong bisimulation S—i.e. it defines Equiv ch to be the largest strong bisimulation. Once again, the decision to develop a polymorphic theory by using a type variable to model names means that the choice function must appear as a parameter to Equiv.

7 The algebraic theory

Having defined strong bisimulation and equivalence in HOL, one may then proceed to develop the algebraic theory presented in [13, 14] as a collection of theorems about the equivalence relation Equiv. Proofs have been completed in HOL for many of the simpler equivalences in this theory, but work on the theory is still in progress. Some examples of the theorems proved so far are the laws for summation shown above in section 2.3, which are expressed in logic by:

$$\vdash \forall ch\ P.\ \mathsf{Equiv}\ ch\ (\mathsf{Plus}\ P\ \mathsf{Zero})\ P$$
$$\vdash \forall ch\ P.\ \mathsf{Equiv}\ ch\ (\mathsf{Plus}\ P\ P)\ P$$
$$\vdash \forall ch\ P\ Q.\ \mathsf{Equiv}\ ch\ (\mathsf{Plus}\ P\ Q)\ (\mathsf{Plus}\ Q\ P)$$
$$\vdash \forall ch\ P\ Q\ R.\ \mathsf{Equiv}\ ch\ (\mathsf{Plus}\ P\ (\mathsf{Plus}\ Q\ R))\ (\mathsf{Plus}\ (\mathsf{Plus}\ P\ Q)\ R)$$

These theorems were proved in HOL in the same way that the corresponding laws are proved in [14], namely by explicitly producing an appropriate strong bisimulation in each case. For example the bisimulation relation used in [14] to prove the commutative law of summation is presented as:

$$\{(P_1 + P_2,\ P_2 + P_1) \mid P_1, P_2 \text{ agents}\} \cup \mathbf{Id}$$

In the HOL proof, the same relation is written

$$\lambda P.\ \lambda Q.\ P{=}Q \lor \exists P'\ Q'.\ P{=}(\mathsf{Plus}\ P'\ Q') \land Q{=}(\mathsf{Plus}\ Q'\ P')$$

The proof that this relation is indeed a strong bisimulation makes extensive use of the theory of the transition system discussed above in section 5.2, as do all the other proofs for the algebraic theory of the π-calculus in HOL.

8 Concluding remarks

This paper has outlined work in progress on a mechanized formal theory of the π-calculus in higher order logic using the HOL system. This theory is still far from complete, and it is still too early to tell if the goals mentioned in the introduction can be achieved. But the results obtained so far seem to indicate that some measure of success is possible. Once the theory is complete, we intend to test it on a realistic application. It would also be interesting to compare the practical utility of the HOL mechanization with a proof system for the π-calculus implemented using a more general logical framework, such as Isabelle or the Edinburgh Logical Framework (LF).

Acknowledgements

I am grateful to Professor Robin Milner for explaining how agent identifiers could be replaced by replication. Thanks are also due to Monica Nesi, who carefully read an early draft of this paper and found several typographical errors, and to Konrad Slind for valuable comments on the theory and its presentation in this paper.

References

[1] R.J.R. Back and J. von Wright, 'Refinement Concepts Formalised in Higher Order Logic', *Formal Aspects of Computing*, Vol. 2, No. 3 (July-Sept. 1990), pp. 247–272.

[2] A.J. Camilleri, 'Mechanizing CSP Trace Theory in Higher Order Logic', *IEEE Transactions on Software Engineering*, Vol. 16, No. 9 (Sept. 1990), pp. 993–1004.

[3] A. Church, 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, Vol. 5 (1940), pp. 56–68.

[4] DSTO and SRI International, *The HOL System: DESCRIPTION* (1991).

[5] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, 'An initial algebra approach to the specification, correctness, and implementation of abstract data types', in *Current Trends in Programming Methodology*, edited by R.T. Yeh (Prentice-Hall, 1978), Vol. IV, pp. 80–149.

[6] M.J.C. Gordon, 'HOL: A Proof Generating System for Higher-Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam (Kluwer Academic Publishers, 1988), pp. 73–128.

[7] M.J.C. Gordon, 'Mechanizing Programming Logics in Higher Order Logic', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam (Springer-Verlag, 1989), pp. 387–439.

[8] M.J. Gordon, A.J. Milner, and C.P. Wadsworth, 'Edinburgh LCF: A Mechanised Logic of Computation', Lecture Notes in Computer Science, Vol. 78 (Springer-Verlag, 1979).

[9] R. Harper, D. MacQueen, and R. Milner, 'Standard ML', Report no. ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (March 1986).

[10] T.F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

[11] T.F. Melham, 'Using Recursive Types to Reason about Hardware in Higher Order Logic', in: *Proceedings of the IFIP WG 10.2 Working Conference on the Fusion of Hardware Design and Verification*, edited by G.J. Milne (North-Holland, 1988), pp. 51–75.

[12] R. Milner, *Communication and Concurrency* (Prentice Hall, 1989).

[13] R. Milner, J. Parrow, and D. Walker, 'A Calculus of Mobile Processes: Part I', Report no. ECS-LFCS-89-85, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (June 1989).

[14] R. Milner, J. Parrow, and D. Walker, 'A Calculus of Mobile Processes: Part II', Report no. ECS-LFCS-89-86, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (June 1989).

[15] L.C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).

Unification of Simply Typed Lambda-Terms as Logic Programming

Dale Miller*
Laboratory for the Foundation of Computer Science
University of Edinburgh, and
Computer Science Department
University of Pennsylvania

March 1991

Abstract

The unification of simply typed λ-terms modulo the rules of β- and η-conversions is often called "higher-order" unification because of the possible presence of variables of functional type. This kind of unification is undecidable in general and if unifiers exist, most general unifiers may not exist. In this paper, we show that such unification problems can be coded as a query of the logic programming language L_λ in a natural and clear fashion. In a sense, the translation only involves explicitly axiomatizing in L_λ the notions of equality and substitution of the simply typed λ-calculus: the rest of the unification process can be viewed as simply an interpreter of L_λ searching for proofs using those axioms.

1 Introduction

Various recent computer systems require typed λ-terms to be unified. For example, the theorem proving systems TPS [1] and Isabelle [14] and the logic programming language λProlog [13] all require unification of simply typed λ-terms. The logic programming language Elf [15], based on the type system LF [5], requires a similar operation for dependent typed λ-terms. Flexible implementations of type systems will probably need to employ various aspects of such unification.

In order to avoid using the very vague and over used adjective "higher-order," we shall refer to the problem of unifying simply typed λ-terms modulo β- and η-conversion as $\beta\eta$-unification. There have been several presentations of $\beta\eta$-unification. One of the first to have been implemented in numerous systems was given by Huet in [7]. Snyder and Gallier in [16] and the author in [11] follow Huet's presentation closely except that details of the search for unifiers are made more declarative using notions similar to the transition systems found in [8].

The presentation given here will depart significantly from those found in these other papers, although interesting connections between these presentations can be made. The most significant departure is that the logic programming language L_λ [10] is employed to assist in specifying

*Appears in the Proceedings of the 1991 International Conference on Logic Programming, edited by Koichi Furukawa, June 1991.

$\beta\eta$-unification. To a certain extent, the transition systems used in [8, 11, 16] could be formalized using first-order Horn clauses. The logic L_λ is more expressive than Horn clauses because it contains constructs for the scoped introduction of program clauses and local constants. These scoping constructs are used to address problems in handling the scopes and names of bound variables in λ-terms.

The logic L_λ is a weak subset of the logic underlying λProlog: it is weaker in that an implementation of L_λ would only need to contain a kind of first-order unification while λProlog needs full higher-order unification. This paper is an attempt to understand exactly this gap and to show that the gap can be bridged completely within the weaker language in a very direct and declarative way.

This paper is divided into the following sections. The next section motivates a style of specification used throughout the paper. Section 3 describes some basic aspects of the simply typed λ-calculus and Section 4 presents two logic programming languages, hh^ω and L_λ, that we use for specification. Equality and substitution are given an L_λ specification in Section 5. The non-deterministic specification of $\beta\eta$-unification is completed in Section 6. Some considerations for producing a deterministic implementation of this specification are given in Section 7. We briefly conclude in Section 8.

2 Motivations

Consider a simple, multi-sorted first-order logic that consists of the primitive types (sorts) $S = \{i, j\}$ and signature (i.e., the set of constants)

$$\Sigma_0 = \{a : i,\ b : j,\ f : i \to j,\ g : j \to i \to i\}.$$

For example, the term $(g\ (f\ a)\ (g\ b\ a))$ is a closed, first-order terms of type i over Σ_0. Let $copy_i$ and $copy_j$ be the binary equality predicates for these two types (the reason for chosing the root word "copy" instead of, say, "equal" will be apparent later). The provable instances of equality for types i and j can be axiomatized using the two clauses

$$\forall_i x.copy_i\ x\ x, \quad \forall_j x.copy_j\ x\ x.$$

(The subscript on a quantifier indicates the type the quantified variable assumes in its scope.) Of course, this description of equality does not provide any information about how equality is checked. It is a convenient specification, however, since it is actually independent of the signature used to build terms of these two sorts. A more detailed specification for these predicates given the signature above would be the clauses $\mathcal{C}_0$ listed below:

$$\begin{gathered} copy_i\ a\ a, \qquad copy_j\ b\ b, \\ \forall_i x \forall_i u(copy_i\ x\ u \supset copy_j\ (f\ x)\ (f\ u)), \\ \forall_j x \forall_j u(copy_j\ x\ u \supset \forall_i y \forall_i v(copy_i\ y\ v \supset copy_i\ (g\ x\ y)\ (g\ u\ v))). \end{gathered}$$

It is a simple matter to prove that if t and s are two closed terms, then $copy_i\ t\ s$ is provable from these formulas if and only if t and s are equal terms of type i over the signature Σ_0. All the clauses in $\mathcal{C}_0$ are essentially first-order Horn clauses.

Given this formulation of equality, it is very simple to specify substitution in the following fashion. Let $x : i$ be a "new" constant (chosen so as not to be in Σ_0), let t be some closed term of type i over Σ_0, and let s be some term over $\Sigma_0 \cup \{x : i\}$. Then it is again an easy matter to show that the atom $copy_i\ s\ r$ is provable from $\mathcal{C}_0$ augmented with the clause $copy_i\ x\ t$ if and only if r is the result of substituting t for x in s; that is, $\mathcal{C}_0 \cup \{copy_i\ x\ t\} \vdash copy_i\ s\ r$ if and only if $r = [x \mapsto t]s$. This simple device of augmenting equality programs will be used frequently to encode substitution. Since *copy* will sometimes indicate equality and sometimes substitution, depending on the context, it was named for a more operational and neutral concept.

Finally, notice that the structure of the signature of Σ_0 gives rise immediately to the structure of the program clauses in $\mathcal{C}_0$. Thus, for each functional arrow $\rightarrow$ in the type of a constant, there corresponds two universal quantifiers and an implication in the program. Following this observation, it seems clear how to incorporate a constant of second order type into the specification of equality. For example, let $\Sigma_1 = \Sigma_0 \cup \{h : (i \rightarrow j) \rightarrow i\}$. The λ-term $h\ (\lambda w.f\ (g\ b\ w))$, for example, is a Σ_1-term of type i. Following the example above, the clause for describing equality for terms containing h should be written as

$$\forall_{i\rightarrow j}x\forall_{i\rightarrow j}u(\forall_i y\forall_i v(copy_i\ y\ v \supset copy_j\ (x\ y)\ (u\ v)) \supset copy_j\ (h\ x)\ (h\ u)).$$

As we shall see later in Section 5, this is the correct axiomatization of equality with respect to the constant h. This clause, however, is clearly not a first-order Horn clause since it contains an implication and universal quantifier in its body and because it uses quantification of the second-order variables x and u.

The material in the next two sections provide a formal background by which the above observations can be made precise and generalized.

3 Simply Typed λ-Calculus

Let S be a fixed, finite set of *primitive types* (also called *sorts*). The set of *types* is the smallest set of expressions that contains the primitive types and is closed under the construction of function types, using 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: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Let τ be the type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $\tau_0 \in S$ and $n \geq 0$. (By convention, if $n = 0$ then τ is simply the type τ_0.) The types $\tau_1, \ldots, \tau_n$ are the *argument types of* τ while the type τ_0 is the *target type of* τ. The order of a type τ is defined as follows: If $\tau \in S$ then τ has order 0; otherwise, the order of τ is one greater than the maximum order of the argument types of τ. Thus, τ has order 1 exactly when τ is of the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau_0$ where $n \geq 1$ and $\{\tau_0, \tau_1, \ldots, \tau_n\} \subseteq S$.

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 (over S)* is a finite set Σ of constants. We often enumerate signatures by listing their members as pairs, written $a\colon\tau$, where a is a constant of type τ. Although attaching a type in this way is redundant, it makes reading signatures easier.

A constant or variable of type τ is a term of type τ. If t is a term of type $\tau \rightarrow \sigma$ and s is a term of type τ, then the application $(t\ s)$ is a term of type σ. Application associates to the left; that is, the expression $(t_1\ t_2\ t_3)$ is read as $((t_1\ t_2)\ t_3)$. Finally, if x is a variable of type τ and t is a term of type σ, then the abstraction $\lambda x\ t$ is a term of type $\tau \rightarrow \sigma$. If Σ is a signature and t is a closed term all of whose constants are members of Σ, then t is a Σ-*term*.

If x and s are terms of the same type then $[x \mapsto s]$ denotes the operation of substituting s for all free occurrences of x, systematically changing bound variables in order to avoid variable capture.

Terms are related to other terms by the following conversion rules.

- The term s α-converts to the term s' if s contains a subformula occurrence of the form $\lambda x\ t$ and s' arises from replacing that subformula occurrence with $\lambda y\ [x \mapsto y]t$, provided y is not free in t.
- The term s β-converts to the term s' if s contains a subformula occurrence of the form $(\lambda x\ t)t'$ and s' arises from replacing that subformula occurrence with $[x \mapsto t']t$.
- The term s η-converts to s' if s contains a subformula occurrence of the form $\lambda x\ (t\ x)$, where x is not free in t, and s' arises from replacing that subformula occurrence with t.

The binary relation of λ-*conversion* is defined so that t λ-converts to s if there is a list of terms $t_1, \ldots, t_n$, with $n \geq 1$, t equal to t_1, s equal to t_n, and for $i = 1, \ldots, n-1$, either t_i converts to t_{i+1} or t_{i+1} converts to t_i by α, β, or η. Expressions of the form $\lambda x\ (t\ x)$ are called η-redexes (provided x is not free in t) while expressions of the form $(\lambda x\ t)s$ are called β-redexes. A term is in λ-normal form if it contains no β- or η-redexes. Every term can be converted to a λ-normal term, and that normal term is unique up to the name of bound variables. The expression $\lambda\mathrm{norm}(t)$ denotes the λ-normal form of t. See [6] for a fuller discussion of these basic properties of the simply typed λ-calculus.

To define formulas, we shall now consider the following extension to terms. Let o be the type of propositions, where o is assumed not to be a member of S. A constant of type $\tau_1 \to \cdots \to \tau_n \to o$ will be used to denote predicates; that is, a predicate is denoted by a functional expression that takes its arguments to a proposition. The logical constants are given the following types: $\wedge, \vee, \supset$ are all of type $o \to o \to o$; and $\forall_\tau$ and $\exists_\tau$ are of type $(\tau \to o) \to o$, for all types τ. We shall rule out quantification over predicates by restricting the type τ in $\forall_\tau$ and $\exists_\tau$ not to contain the type symbol o. We shall assume that the logical constants are not members of any signature. A formula is a term of type o. The logical constants $\wedge, \vee, \supset$ are written in the familiar infix form. The expressions $\forall_\tau(\lambda z\ t)$ and $\exists_\tau(\lambda z\ t)$ are written simply as $\forall_\tau z\ t$ and $\exists_\tau z\ t$. A closed formula is a Σ-*formula* if all of its non-logical constants are members of Σ. The substitution operation and conversion relations on terms immediately extend to formulas.

4 Two Logic Programming Languages

4.1 Hereditary Harrop formulas: hh^ω

Our first logic programming language, called hh^ω, is based on two sets of closed, λ-normal formulas: $\mathcal{D}$, which can be used as program clauses, and $\mathcal{G}$, which can be used as goals or queries. The formulas in $\mathcal{D}$, denoted by the syntactic variable D, are those that do not have any positive occurrence of a disjunction or existential quantifier, while formulas in $\mathcal{G}$, denoted by the syntactic variable G, are their dual; that is, formulas in $\mathcal{G}$ cannot have any negative occurrence of a disjunction or existential quantifier.

In order to formalize a notion of backchaining over clauses of this general form, we need the following definition. Let $\mathcal{P}$ be a finite subset of $\mathcal{D}$. The set of pairs $|\mathcal{P}|_\Sigma$ is defined to be the smallest set such that

- if $D \in \mathcal{P}$ then $\langle \emptyset, D\rangle \in |\mathcal{P}|_\Sigma$,
- if $\langle \Gamma, D_1 \wedge D_2\rangle \in |\mathcal{P}|_\Sigma$ then $\langle \Gamma, D_1\rangle$ and $\langle \Gamma, D_2\rangle$ are members of $|\mathcal{P}|_\Sigma$,

- if $\langle \Gamma, G \supset D\rangle \in |\mathcal{P}|_\Sigma$ then $\langle \Gamma \cup \{G\}, D\rangle \in |\mathcal{P}|_\Sigma$, and
- if $\langle \Gamma, \forall_\tau x\, D\rangle \in |\mathcal{P}|_\Sigma$ and t is a Σ-term, then $\langle \Gamma, \lambda\mathrm{norm}([x \mapsto t]D)\rangle \in |\mathcal{P}|_\Sigma$.

The following proposition has been used elsewhere to justify calling hh^ω a logic programming language [9, 12].

Proposition 1 *Let Σ be a signature, let $\mathcal{P}$ be a finite subset of $\mathcal{D}$, let $\{G_1, G_2, \exists_\tau x.G, \forall_\tau x.G\} \subseteq \mathcal{G}$, and let $D \in \mathcal{D}$. Then the following holds for intuitionistic provability $\vdash$. (When we write $\Sigma; \mathcal{P} \vdash G$ we assume that $\mathcal{P} \cup \{G\}$ is a set of Σ-formulas.)*

- $\Sigma; \mathcal{P} \vdash G_1 \wedge G_2$ *if and only if* $\Sigma; \mathcal{P} \vdash G_1$ *and* $\Sigma; \mathcal{P} \vdash G_2$.
- $\Sigma; \mathcal{P} \vdash G_1 \vee G_2$ *if and only if* $\Sigma; \mathcal{P} \vdash G_1$ *or* $\Sigma; \mathcal{P} \vdash G_2$.
- $\Sigma; \mathcal{P} \vdash \exists_\tau x\, G$ *if and only if there is a Σ-term t of type τ such that* $\Sigma; \mathcal{P} \vdash \lambda norm([x \mapsto t]G)$.
- $\Sigma; \mathcal{P} \vdash D \supset G_1$ *if and only if* $\Sigma; \mathcal{P} \cup \{D\} \vdash G_1$.
- $\Sigma; \mathcal{P} \vdash \forall_\tau x.G$ *if and only if* $\Sigma \cup \{c : \tau\}; \mathcal{P} \vdash [x \mapsto c]G$*, where c is a constant of type τ that is not in Σ.*
- *If A is atomic, then $\Sigma; \mathcal{P} \vdash A$ if and only if for some Γ, $\langle \Gamma, A\rangle \in |\mathcal{P}|_\Sigma$ and for every $G \in \Gamma$, $\Sigma; \mathcal{P} \vdash G$.*

This proposition in fact describes a non-deterministic interpreter for hh^ω. Moving from this proposition to an actual deterministic interpreter for hh^ω is a difficult task. Various aspects of implementing a language like hh^ω have been considered in [3, 13]. We mention a couple aspects in Section 7. In order to motivate introducing the next logic programming language, it is important to mention here that an interpreter for hh^ω will need to perform β-reductions while looking for proofs. That is, although programs and goals start out in λ-normal form, substitutions may cause them to become non-normal. Thus, references to the λnorm() function in Proposition 1 and in the definition of $|\mathcal{P}|_\Sigma$ are necessary in general. It is because β-conversion can cause significant changes to a term that unification in this setting is very hard. The next language we introduce will be restricted in such a way that only a very simple fragment of general β-conversion is required in the interpreter. As a result, unification in that language will be particularly simple.

4.2 The sublanguage: L_λ

A bound variable occurrence in a formula $G \in \mathcal{G}$ is *essentially universal* if it is bound by a positive occurrence of a universal quantifier or by a (term-level) λ-abstraction in G; it is *essentially existential* if it is bound by either a positive existential or a negative universal quantifier in G. Dually: a bound variable occurrence in a formula $D \in \mathcal{D}$ is *essentially existential* if it is bound by a positive occurrence of a universal or negative occurrence of an existential quantifier in D; it is *essentially universal* if it is bound by a negative universal quantifier or a (term-level) λ-abstraction in D. In the running of the non-deterministic interpreter described above, the essentially existential variables can get instantiated with general terms; it is via substitutions for these variables that new β-redexes can appear.

Our second logic programming language, called L_λ, is based on two sets of λ-normal formulas: $\mathcal{D}' \subseteq \mathcal{D}$, which can be used as program clauses, and $\mathcal{G}' \subseteq \mathcal{G}$, which can be used as goals or queries. The restriction to determine the subsets intended is the following in both cases:

whenever any formula in these sets has an essentially existential bound variable occurrence, say x, appearing as the head of an expression of the form $(x\ t_1\ \ldots\ t_n)$ $(n \geq 1)$ then $t_1, \ldots, t_n$ is a list of distinct variables that are essentially universally quantified within the scope of the binding for x.

For example, if predicate p has type $j \to o$ then the formula

$$\forall_{i\to j} x \forall_i y (p\ (x\ y) \supset p\ (f\ y))$$

is an example of both a goal and program clause for hh^ω; it is only a legal goal in L_λ. As a clause of L_λ, it has a subterm occurrence $(x\ y)$ where both x and y are essentially existential. Such a subterm is not premitted. All the formulas in Section 2 are in L_λ. Of course, first-order Horn clauses are both goals and clauses in L_λ.

Given this restriction to the syntax of program clauses and goals, the only β-redexes that must be computed from within an interpreter for L_λ are those of the form $(\lambda x.M)y$ where y is a bound variable that is not free in $\lambda x.M$. Such β-redexes are very simple to reduce: just change free occurrences of x in M to y. Given that α-conversion is available, this can be stated even more simply: a term t is related to s by β_0*-conversion* if one is gotten from the other by replacing a β_0*-redex* $(\lambda x.M)x$ in one with M in the other. If the interpreter for hh^ω is given a program and goal of the restricted language L_λ, the only β-redexes that need to be reduced are β_0-redexes.

It is proved in [10] that the unification problems that arise from writing an interpreter for L_λ, say $\beta_0\eta$-unification problems, are decidable and most general unifiers exists when unifiers exist. It is argued in that paper that the unification needed for L_λ is the weakest extensions to first-order unification that treats bound variables directly.

5 Specifying Equality and Substitution

Let t and s be two λ-normal terms of type σ. Define the following function by induction on the structure of simple types.

$$[\![t, s : \sigma]\!] = \begin{cases} \lambda\mathrm{norm}(copy_\sigma\ t\ s) & \text{if } \sigma \text{ is primitive} \\ \forall_{\sigma_1} x \forall_{\sigma_1} u([\![x, u : \sigma_1]\!] \supset [\![(t\ x), (s\ u) : \sigma_2]\!]) & \text{if } \sigma \text{ is } \sigma_1 \to \sigma_2. \end{cases}$$

(This recursive definition is similar to that used in [4] to code a dependent typed λ-calculus into hh^ω.) For example, the expression $[\![h, \lambda x.(g\ (x\ a)\ a) : (i \to j) \to i]\!]$ yields the formula

$$\forall_{i\to j} x \forall_{i\to j} u (\forall_i y \forall_i v [copy_i\ y\ v \supset copy_j\ (x\ y)\ (u\ v)] \supset copy_i\ (h\ x)\ (g\ (u\ a)\ a)).$$

Notice that the clauses given in Section 2 are exactly the clauses

$$[\![a, a : i]\!],\ [\![b, b : j]\!],\ [\![f, f : i \to j]\!],\ [\![g, g : j \to i \to i]\!],\ [\![h, h : (i \to j) \to i]\!]$$

It is an easy matter to show that such a formula is always both a goal and a clause for L_λ and that the formula $[\![t, s : \sigma]\!]$ is a Horn clause if and only if σ is of order 0 or 1. The following proposition is stated here without proof. Its proof is a straightforward induction on the structure of proofs (which mirrors the structure of $\beta\eta$-long normal forms [6]).

Proposition 2 *Let Σ contain at least the distinct constants $c_1 : \sigma_1, \ldots, c_n : \sigma_n$ $(n \geq 0)$. Let $t_1, \ldots, t_n$ be Σ-terms of type $\sigma_1, \ldots, \sigma_n$, respectively, and let $\mathcal{C}$ be the set $\{[\![c_i, t_i : \sigma_i]\!] \mid i = 1, \ldots, n\}$. Finally, let M and N be Σ-terms of type τ. Then $\Sigma; \mathcal{C} \vdash [\![M, N : \tau]\!]$ if and only if M is a $\{c_1, \ldots, c_n\}$-term and $(\lambda c_1 \ldots \lambda c_n.M)t_1 \ldots t_n$ $\beta\eta$-converts to N.*

From this proposition, the following corollary follows immediately.

Corollary 1 *Let Σ be a signature and let $\mathcal{C}_\Sigma$ be the set $\{[\![c, c : \sigma]\!] \mid c : \sigma \in \Sigma\}$.*

- *If M and N are Σ-terms of type τ, then $\Sigma; \mathcal{C}_\Sigma \vdash [\![M, N : \tau]\!]$ if and only if M $\beta\eta$-converts to N.*
- *If M and N are Σ-terms of type $\sigma \to \tau$ and τ, respectively, then $\Sigma; \mathcal{C}_\Sigma \vdash \forall_\sigma x([\![x, t : \sigma]\!] \supset [\![M\ x, N : \tau]\!])$ if and only if $(M\ t)$ $\beta\eta$-converts to N.*

Thus, it is possible to use the formulas $[\![c, c : \sigma]\!]$ to help specify both equality (that is, $\beta\eta$-conversion) and substitution. To illustrate how substitution can be axiomatized, consider the following L_λ clause

$$\forall_i x(copy_i\ x\ T \supset copy_i\ (M\ x)\ S) \supset subst_{i\to i}\ M\ T\ S.$$

(Here, we shall start adopting the (familiar Prolog) convention that essentially existential variables will be capitalized letters and that if any variable is not explicitly quantified, it is assumed to be universally quantified or existentially quantified with outermost scope depending on whether or not the formula is intended to be a program clause or a goal.) The type of $subst_{i\to i}$ is $(i \to i) \to i \to i \to o$.

Assume for the moment that we have a Prolog-like interpreter for L_λ, and consider attempting to find a substitution term for the (essentially existential) variable F so that the goal $subst_{i\to i}\ F\ a\ (g\ b\ a)$ is provable from this clause and the clauses in Section 1. Backchaining would cause this goal to be reduced to

$$\forall_i x(copy_i\ x\ a \supset copy_i\ (F\ x)\ (g\ b\ a)).$$

This goal is then reduced by introducing a new constant, say $c : i$, and then adding the clause $copy_i\ c\ a$ to the program before attempting to prove the goal $copy_i\ (F\ c)\ (g\ b\ a)$. Notice that since c was introduced after the "logic" variable F was introduced, a correct interpreter for L_λ would need to make certain that F is not instantiated with a term that contains c. There is only one clause, namely $[\![g, g : j \to i \to i]\!]$, on which to backchain to prove this goal. Doing so reduces this goal to the two goals $copy_j\ F_1\ b$ and $copy_i\ F_2\ a$, where the disagreement pair $F\ c = g\ F_1\ F_2$ must still be solved. The first of these two goals has exactly one solution, namely $F_1 \mapsto b$, gotten by backchaining on $copy_j\ b\ b$. The second goal, however, can be proved two different ways: by backchaining over either $copy_i\ a\ a$, yielding $F_2 \mapsto a$ or $copy_i\ c\ a$ yielding $F_2 \mapsto c$. Putting these substitutions back together, we get two different solutions to the original goal: namely $F \mapsto \lambda w.(g\ b\ a)$ by solving the disagreement pair $F\ c = g\ b\ a$, and $F \mapsto \lambda w.(g\ b\ w)$ by solving the disagreement pair $F\ c = g\ b\ c$. (The possible solution $F \mapsto \lambda w.g\ b\ c$ is ruled out since c is not permitted to occur free in the substitution term of F.) Notice, that these two substitutions are exactly the two solutions to the $\beta\eta$-unification problem $\exists_{i\to i}F.\ F\ a = g\ b\ a$.

Substitution can be axiomatized in a general fashion by extending the example above. Assume that we have the predicates

$$subst_{\tau\to\sigma} : (\tau \to \sigma) \to \tau \to \sigma \to o$$

for each pair of types τ and σ. These predicates are then axiomatized by the following clause scheme:

$$\forall_\tau x([\![x, T : \tau]\!] \supset [\![(M\ x), S : \sigma]\!]) \supset subst_{\tau\to\sigma}\ M\ T\ S.$$

It follows immediately from the corollary above, that, when used in conjunction with the clauses $\{[\![c, c : \sigma]\!] \mid c : \sigma \in \Sigma\}$, these clauses prove $(subst_{\tau\to\sigma}\ M\ T\ S)$ if and only if $(M\ T)$ is $\beta\eta$-convertible to S. Now assume that M is of the form $\lambda x.M'$. The computation of the λ-normal form of $(M\ T)$ happens in two steps. First, T is substituted for free occurrences of x in M'. It is this step that the logical structure of the *subst* clause makes explicit. The second step requires any newly introduced β-redexes to be reduced. This step is not made explicit in the code above: if it needs to happen, the meta-level proof operation must perform those reductions. Thus, the *subst*-clauses cannot generally be L_λ program clauses. For example, the clause specifying substitution at type $(i \to j) \to i$ is

$$\forall_{i\to j}x(\forall_i Y \forall_i V(copy_i\ Y\ V \supset copy_j\ (x\ Y)\ (T\ V)) \supset copy_i\ (M\ x)\ S) \supset subst_{(i\to j)\to i}\ M\ T\ S.$$

This clause is not an L_λ program clause because the essentially existential variable T has an occurrence $(T\ V)$ where it is applied to another essentially existential variable. New β-redexes (which are not β_0-redexes) can be introduced at this point. In defining $subst_{\tau\to\sigma}$, if the type τ is primitive, then no new β-redexes will appear and the corresponding *subst* clause is, in fact, in L_λ (see the axiomatization of $subst_{i\to i}$ above).

It is possible to axiomatize *subst* completely in L_λ. Since it can be determined statically where β-reductions will need to be performed within the computation of a *subst* goal, it is possible to replace the β-redex with an explicit call to *subst*, this time at a lower type. In particular, if τ is functional, then $subst_{\tau\to\sigma}$ will need to call $subst_\tau$. For example, the following is an L_λ specification of $subst_{(i\to j)\to i}$:

$$\begin{aligned}\forall_{i\to j}x(\forall_i Y \forall_i V(copy_i\ Y\ V \supset \forall_j U[subst_{i\to j}\ T\ V\ U \supset copy_j\ (x\ Y)\ U])\\ \supset copy_i\ (M\ x)\ S) \supset subst_{(i\to j)\to i}\ M\ T\ S.\end{aligned}$$

Here, the positively occurring atom $copy_j\ (x\ Y)\ (T\ V)$ is replaced with

$$\forall_j U[subst_{i\to j}\ T\ V\ U \supset copy_j\ (x\ Y)\ U]:$$

the β-reduction needed to simplify $(T\ V)$ is made explicit by the call to $subst_{i\to j}$.

The two implementations of *subst* prove the same goals. In this fashion, we shall assume that the predicates $subst_{\tau\to\sigma}$ are all axiomatized completely in L_λ. The translation of a clause of hh^ω into a clause in L_λ given by this example can be generalized. We present a general translation in the next section.

6 Transforming hh^ω Goals into L_λ Goals

It is possible to systematically translate a goal in hh^ω into a goal in L_λ so that the proofs of the goal in hh^ω differ from the proofs in L_λ only in that additional *subst* and *copy* goals need to be established. Otherwise, all substitutions made in these proofs are identical. Since $\Sigma; \{D_1, \ldots, D_n\} \vdash G$ is equivalent to $\Sigma; \emptyset \vdash D_1 \supset \cdots \supset D_n \supset G$, it is enough to restrict this translation to goal formulas only: it dualizes immediately for program clauses.

For convenience, we axiomatize the predicates

$$subst^n_{\tau_1\to\cdots\to\tau_n\to\sigma} : (\tau_1 \to \cdots \to \tau_n \to \sigma) \to \tau_1 \to \cdots \to \tau_n \to \sigma \to o$$

to do an n-fold substitution in the following way:

$$\begin{aligned}\forall_{\tau_1}x_1([\![x_1, T_1 : \tau_1]\!] \supset \cdots \supset \forall_{\tau_n}x_n([\![x_n, T_n : \tau_n]\!] \supset [\![Mx_1 \cdots x_n, S : \sigma]\!]) \ldots)\\ \supset subst^n\ M\ T_1 \ldots T_n\ S.\end{aligned}$$

As before, these clauses can be adjusted so that they are actually L_λ program clauses.

A subterm of a goal formula G will be called a non-L_λ subterm if it is of the form $(X\ t_1 \cdots t_n)$ where $n \geq 1$, X is essentially existential in G, and the terms $t_1, \ldots, t_n$ do not satisfy the restrictions defining L_λ. The translation from hh^ω goals to L_λ goals is by induction on the number of non-L_λ subterm occurrences. Let $G \in \mathcal{G}$. If G is not an L_λ goal, then there is an occurrence of an atomic formula A in G which has a subterm of the form $(X\ t_1 \cdots t_n)$ where n, X and $t_1, \ldots, t_n$ are as above. Let $y_1, \ldots, y_m$ $(m \geq 0)$ be the list of essentially universal variables that are bound in the scope of X's binding occurrence and that also contain $(X\ t_1 \cdots t_n)$ in their scope. Let $\sigma_1, \ldots, \sigma_m$ be the types of $y_1, \ldots, y_m$, respectively. Let H be a variable not occurring free in A and let A' be the result of replacing the occurrence of $(X\ t_1 \cdots t_n)$ with $(H\ y_1 \ldots y_m)$. Let G' be the result of replacing A with either the expression

$$\forall H([\forall_{\sigma_1} y_1(\llbracket y_1, y_1 : \sigma_1 \rrbracket \supset \cdots \supset \forall_{\sigma_m} y_m(\llbracket y_m, y_m : \sigma_m \rrbracket \supset \\ subst^n\ X\ t_1 \ldots t_n\ (H\ y_1 \ldots y_m)) \cdots)] \supset A')$$

if A occurs negatively in G, or

$$\exists H([\forall_{\sigma_1} y_1(\llbracket y_1, y_1 : \sigma_1 \rrbracket \supset \cdots \supset \forall_{\sigma_m} y_m(\llbracket y_m, y_m : \sigma_m \rrbracket \supset \\ subst^n\ X\ t_1 \cdots t_n\ (H\ y_1 \ldots y_m)) \ldots)] \wedge A')$$

if A occurs positively in G. The resulting formula now has one fewer non-L_λ subterms. If we repeat this process until all such subterms are removed, the result will be an L_λ goal formula. The following proposition establishes a simple correctness property for this translation.

Proposition 3 *Let Σ be a signature, let $\mathcal{C}_\Sigma$ be the set $\{\llbracket c, c : \sigma \rrbracket \mid c : \sigma \in \Sigma\}$, and let G be an hh^ω goal formula that does not contain occurrences of any* copy *or* subst *predicates. Let G'' be the L_λ goal that results from performing the above mentioned translations to the hh^ω goal formula G. Then*

$$\Sigma; \emptyset \vdash G \quad \textit{if and only if} \quad \Sigma; \mathcal{C}_\Sigma \vdash G''.$$

Furthermore, if G is of the form $\exists_\tau X.H$ then G'' is of the form $\exists_\tau X.H''$, and for all Σ-terms t of type τ,

$$\Sigma; \emptyset \vdash \lambda norm([X \mapsto t]H) \quad \textit{if and only if} \quad \Sigma; \mathcal{C}_\Sigma \vdash \lambda norm([X \mapsto t]H'').$$

In other words, answers substitutions are the same between these two goals.

Consider the $\beta\eta$-unification problem

$$\exists_{i \to i} X[k(\lambda v(m\ (X\ v))) = k(\lambda y(X\ (m\ y)))],$$

over the signature $\{m : i \to i,\ k : (i \to i) \to i\}$. This can be written as the hh^ω goal

$$\forall_i Z(eq\ Z\ Z) \supset \exists_{i \to i} X[eq\ (k(\lambda v(m\ (X\ v))))\ (k(\lambda y(X\ (m\ y))))].$$

This query has one non-L_λ subterm, namely, $(X\ (m\ y))$. Using the above mentioned transformation leads to the L_λ goal

$$\forall_i Z(eq\ Z\ Z) \supset \exists_{i \to i} X \exists_{i \to i} H[\forall_i y(copy_i\ y\ y \supset subst_{i \to i}\ X\ (m\ y)\ (H\ y)) \wedge \\ eq\ (k(\lambda v(m\ (X\ v))))\ (k(\lambda y(H\ y)))].$$

Solving this query using the *copy*-clauses for the constants m and k, we find that there are an infinite number of proofs, yielding a sequence of substitution terms for X, namely, $\lambda w.w$, $\lambda w.(f\ w)$, $\lambda w.(f(f\ w))$, etc. These are thus the unifiers for the original $\beta\eta$-unification problem.

7 Some Implementation Considerations

The presentation of $\beta\eta$-unification given so far is rather simple and declarative. This is partly due to pushing lots of details concerning bound variables and search into an imaginary interpreter for L_λ, where, presumably, all these details must be carefully addressed. The eLP implementation of λProlog [2] provides an interpreter for both hh^ω and L_λ. Many details regarding how such an interperter can be built are given in the paper [3].

There are only two special implementation considerations that we would like to address here. Assume that we are designing an interpreter that uses logic variables and unification in the usual way to postpone determining substitution terms for essentially existential variables. Let t be a λ-normal term of primitive type. We say t is *flexible* if its head symbol is a logic variable; otherwise, it is *rigid*. If σ is a primitive type, a goal of the form $copy_\sigma\ t\ s$ is classified as a rigid-rigid, flexible-rigid, rigid-flexible, or flexible-flexible *copy*-goal depending on the status of the two terms t and s. Given that *copy* is axiomatized only in the forms used in this paper, that is, as axiomatizations of equality and substitution, then we can conclude the following behavior for proving *copy*-goals. A rigid-rigid or rigid-flexible *copy*-goal can be used to backchain over at most one clause. These are therefore "deterministic" goals. A flexible-rigid goal may have several clauses to backchain over. Consider the case where the top-level constant of s is, say, $f : i \rightarrow i$. Thus, it will be possible to backchain using the clause $[\![f, f : i \rightarrow i]\!]$. There may have also been extensions to the program via *subst*-goals in which clauses of the form $copy_i\ x\ (f\ u)$, where x is some "new" constant. It is then possible to backchain over these additional clauses. Thus, flexible-rigid *copy*-goals give rise to the searching that goes on in $\beta\eta$-unification: it is this non-determinancy that is reflected in the MATCH procedure of [7]. A sensible interpretation of such a goal is to order these backchaining choices and to try one after the other using some search discipline (such as depth-first search).

Consider proving a flexible-flexible goal $copy_\sigma\ t\ s$. Every $copy_\sigma$-clause in the program could be used to backchain on this goal: there may be a large number of such clauses. Also, backchaining over a clause encoding a constant of non-primitive type will generate more flexible-flexible *copy*-goals and these may also be used to backchain over a large number of clauses. In certain cases, say when the only clauses for $copy_\sigma$ are for constants that are of primitive type or when there are, in fact, no clauses for $copy_\sigma$ (σ is an empty type), proving such flexible-flexible *copy*-goals may not lead to an explosion in the search space. In general, however, a sensible approach to proving flexible-flexible goals would be to suspend them (using delay mechanisms such as in NU-Prolog [17]) and attempt to prove other goals in the hope that the flexible heads will be instantiated to make them rigid, at which point they could be resumed. Suspending such flexible-flexible *copy*-goals is similar to advice given in [7] and to the treatment of $\beta\eta$-unification in λProlog [13] and its current implementations [2, 3].

As a final comment on an implementation, consider the occurrence-check in the usual first-order unification algorithm. This does not generalize directly when variables of higher-order type are present. For example, the unification problem $\exists_i X \exists_{i\rightarrow i} F.\ X = F\ X$ has answer substitutions as long as there are terms of type i. For example, if a is of type i, then set F to $\lambda w.w$ or to $\lambda w.a$ and set X to a. In our setting, however, it is possible to generalize the occurrence-check by a check on ancestor goals. That is, if the current *copy*-goal is subsumed by an ancestor goal, then it is possible to fail the current goal without loss of completeness. In the first order setting where the clauses $[\![t, s : \sigma]\!]$ are just Horn clauses, this is equivalent to the occurrence-check. In the general case, however, the ancestor check is still a legitimate step although a simple occurrence-check is not. Thus, a reasonable implementation of L_λ when employed to deal with $\beta\eta$-unification might well have such an ancestor check to stop this simple kind of infinite branch. For more explicit information on generalizations of the occurrence-check to $\beta\eta$-unification, see [7, 11].

8 Conclusion

We have presented a specification of $\beta\eta$-unification using the logic programming language L_λ. This specification approach simplifies the presentation of an implementation of $\beta\eta$-unification by allowing us to focus on the simple declarative aspects of equality and substitution in isolation from details of search and the complex, low-level syntax of λ-abstractions. These latter details are addressed by an implementation of L_λ. Fortunately, a great many techniques and ideas from the implementation of various other logic programming languages can be used to build such an implementation. Of course, L_λ is of greater interest than as just the basis for implementing $\beta\eta$-unification. The translation given in Section 6 shows that many λProlog programs can be translated to L_λ programs in a very direct fashion. Thus, an L_λ interpreter could be used as the core of a λProlog interpreter.

9 Acknowledgements

I am grateful to Bob Constable, Fernando Pereira, and the conference reviewers for their comments and suggestions on an earlier draft of this paper. At the University of Edinburgh, this work has been supported by SERC Grant No. GR/E 78487 "The Logical Framework" and ESPRIT Basic Research Action No. 3245 "Logical Frameworks: Design, Implementation, and Experiment." At the University of Pennsylvania, from where the author is on a one year leave, this work has been supported by ONR N00014-88-K-0633 and NSF CCR-87-05596.

References

[1] Peter B. Andrews, Eve Longini Cohen, Dale Miller, and Frank Pfenning. Automating higher order logic. In *Automated Theorem Proving: After 25 Years*, pages 169–192. American Mathematical Society, 1984.

[2] Conal Elliott and Frank Pfenning. eLP, a Common Lisp Implementation of λProlog. Implemented as part of the CMU ERGO project, May 1989.

[3] Conal Elliott and Frank Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation.* MIT Press, 1990. (in press).

[4] Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449, pages 221–235. Springer Lecture Notes in Artificial Intelligence, 1990.

[5] 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.

[6] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinatory Logic and Lambda Calculus*. Cambridge University Press, 1986.

[7] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[8] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Lanuages and Systems*, 4(2):258–282, April 1982.

[9] Dale Miller. Abstractions in logic programming. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 329 – 359. Academic Press, 1990.

[10] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Extensions of Logic Programming*. Springer Lecture Notes in Artificial Intelligence, 1990.

[11] Dale Miller. Unification under a mixed prefix. To appear in the *Journal of Symbolic Computation*.

[12] 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*.

[13] Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[14] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In Peirgiorgio Odifreddi, editor, *Logic and Computer Science*, pages 361 – 386. Academic Press, 1990.

[15] 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.

[16] Wayne Snyder and Jean H. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1–2):101–140, 1989.

[17] James Thom and Justin Zobel. NU-Prolog reference manual, version 1.3. Technical report, Department of Computer Science, University of Melbourne, Australia, 1988.

]

Finding the Answers in Classical Proofs: A Unifying Framework

Chet Murthy
Cornell University

June 1991

Abstract

Recently there has been much interest in the problem of finding the computational content of classical reasoning. Work in this area has for the most part focused on re-interpreting various classical results [Fri78, Sza, God65, Pra81]. These different initial starting points result in (ostensibly) completely different notions of computation, spanning the range from concurrent, to sequential algorithms, and yielding semantics of valuations, finite domains, and coherence spaces. In this paper, we attempt to provide a unifying framework for viewing these diverse attempts, showing that work of Murthy, and Barbanera and Berardi can be smoothly integrated into the same framework, and seen as different facets of the same phenomenon. We make the first steps toward integrating Girard's classical interpretation into this framework, in the process discovering a new classical reduction, computing under λ-abstraction, which must be added in order to represent his system. We argue that the computational content of a classical proof is most simply understood as a program in an almost-functional language (enriched with Felleisen's "control" operator, $\mathcal{C}$), and that this interpretation yields, under various restrictions, the two different methods of extracting computations from classical proofs we examine.

1 Introduction

Since the original rift between classical and constructive systems, there has been much research and philosophical discussion about the extent to which a classical proof is already constructive. However, unlike the case for constructive systems, there is no Curry-Howard-like isomorphism to guide us in our attempts to find the constructive or computational content of classical proofs. There have been, of course, various interpretations of classical systems into their constructive subsystems, or into other constructive systems [God65, Kol67, chetGir87], and attempts to provide cut-elimination procedures for classical logics [Sza, Pra65]. But there has been only partial success in the endeavour of identifying in a complete manner the extent to which classical proofs encode computations, and in some cases, correct computations.

In this paper, we intend to present a candidate for such a complete account (for classical natural deduction). Our work stems from Friedman's conservative extension of (suitable) classical over constructive systems for Π_2^0 sentences. In [Mur91b] we demonstrated that one could use this result to extract sound programs (i.e. that compute evidence in the constructive sense [How80, chetCon71, Bis70]) directly from classical proofs, in the programming language of call-by-name or call-by-value $\lambda + \mathcal{C}$.

Here, we wish to build upon this account, and view the language of $\lambda + \mathcal{C}$ as the true computational content of classical proofs. We will demonstrate that we can give types to programs in $\lambda + \mathcal{C}$ in such a manner that any reduction step preserves typing. This yields a non-confluent, but well-typed, reduction system, and we will argue that the true computational content of classical proofs is multi-valued.

Based on this foundation, we will examine the work of Berardi and Barbanera, and show that it can be interpreted as employing a particular subset of the reduction rules for $\lambda + \mathcal{C}$, and that in fact this is a complete characterization of their computation system.

We then turn to Girard's translation [Gir91] and find that in fact Felleisen's reduction system is not adequate. It turns out that Girard's translation evaluates under λ-abstractions, and it is for this reason that Felleisen's rules (which all assume that λ-abstractions are values) do not suffice.

Let us begin by fixing the domain of discourse. We will use LJ to refer to the propositional intuitionistic sequent calculus, multi-hypothesis, single/null conclusion. By LJ+$\neg\neg$-elim, we will mean LJ with the rule of double-negation elimination $(\neg\neg(P) \vdash P)$. We will also write this NK, since NK proofs have the same relation to LJ+$\neg\neg$-elim proofs as NJ proofs have to LJ proofs.

1.1 Preliminaries

In this paper we assume familiarity with the concepts discussed in the seminal [Plo75], as well as [FFED86]. In addition, we recommend that the reader have available one or more of [Gri90, Mur91b, Mur91a]. Finally, we assume familiarity with the Nuprl term language and computation system [Con86], or, equivalently, Martin-Löf type theory [ML82]. For reasons of space, we limit discussion of the matters covered in these papers to a minimum, instead giving pointers to the relevant literature.

In the following, we will use $E[]$ to denote evaluation contexts, in the manner of [FF86], and $C[]$ to denote arbitrary term-contexts, which may bind free variables.

2 An Intuitive Discussion of Computations in Classical Proofs

An intuitive understanding of why there should be computations in classical proofs begins with a very simple example. Let us prove that there is a prime number:

$\vdash\ \exists x \in \mathsf{N}.prime(x)$ `BY ¬¬-elim`
  $\vdash\ \neg\neg(\exists x \in \mathsf{N}.prime(x))$ `BY intro` k
    $k{:}\neg(\exists x \in \mathsf{N}.prime(x))\ \vdash\ \bot$ `BY function-elim` k
      $\vdash\ \exists x \in \mathsf{N}.prime(x)$ `BY intro 37`

It is quite clear that this proof contains within it the *evidence* for the proposition $\exists x \in \mathsf{N}.prime(x)$. But if we believe that there is no constructively intelligible evidence in classical proofs, then we are committed to the view that this proof computes nothing. After all, the rule of $\neg\neg$-elim is the classical axiom. We can find a slightly more complicated example, which sheds a bit more light on the subject (let $\varphi \equiv \exists x \in \mathsf{N}.prime(x)$, and FLT be Fermat's Last Theorem, or any arbitrary formula):

$\vdash \exists x \in \mathsf{N}.prime(x)$ BY $\neg\neg$-elim THEN intro k
  $k{:}\neg(\varphi) \vdash \bot$ BY elim k
    $\vdash \varphi$ BY cut $f{:}FLT^1 \Rightarrow \varphi$
      $\vdash FLT \Rightarrow \varphi$ BY intro g
        $g{:}FLT \vdash \varphi$ BY intro 5 & proof of $prime(5)$
      $f{:}FLT \Rightarrow \varphi \vdash \varphi$ BY elim f
        $\vdash FLT$ BY $\neg\neg$-elim
          $\vdash \neg\neg(FLT)$ BY intro h
            $h{:}\neg(FLT) \vdash \bot$ BY elim k
              $\vdash \varphi$ by intro 9 & proof of $prime(9)$

It should be clear that there are two different prime numbers in this proof, and that they are radically different. And we might want our computational interpretation of classical proofs to yield both answers, rather than simply yielding one. We close with a simple example of a proof in which we cannot find evidence, because it is clear none could exist. We will refer to this as the "minimum" theorem: every infinite sequence of bits has a minimum (let $\psi \equiv \exists n \in \mathsf{N}.\forall m \in \mathsf{N}.f(n) \leq f(m)$):

$\vdash \psi$ BY double-negation elim
 $\neg(\psi) \vdash \bot$ BY function-elim
  $\vdash \psi$ BY intro 0
   $\vdash \forall m \in \mathsf{N}.f(0) \leq f(m)$ BY function intro
    $m : \mathsf{N} \vdash f(0) \leq f(m)$ BY cases on $f(0) \leq f(m)$
      $f(0) \leq f(m) \vdash f(0) \leq f(m)$ BY hypothesis
      $f(m) < f(0) \vdash f(0) \leq f(m)$
       BY double-negation elim
          THEN function-elim
        $f(m) < f(0) \vdash \psi$ BY intro m, etc

Clearly, this proof contains no computations, since there can be no function which computes the desired witness, "n".

3 A First Attempt

A first, but incomplete, means of obtaining computations from classical proofs is to generalize Friedman's metatheorem [Fri78] which shows that suitable classical systems conservatively extend their constructive subtheories for Σ_1^0 sentences. His metatheorem provides a method of translating a classical proof of a Σ_1^0 sentence into a constructive proof, from which standard techniques suffice to extract the computation. These results are more fully explained in [Gri90, Mur90, Mur91b]; we summarize here, specializing to classical arithmetic.

Theorem 1 (Conservative Extension (Friedman)) *Given a classical proof of a Σ_1^0 sentence, $\vdash_{PA} \exists x.\ f(x) = 0$:*

1. *with double-negation translation [God65, Kol67], we can show $\vdash_{HA} \neg\neg(\exists x.\ \neg\neg(f(x) = 0))$, without use of $\bot$-elimination.*
2. *We may replace instances of $\bot$ in that proof with an arbitrary closed proposition A, in particular, $\exists x.\ f(x) = 0$, and this A-translated sentence is also provable in HA.*
3. *From this A-translated sentence we can arrive at a constructive proof of the original sentence, $\vdash_{HA} \exists x.\ f(x) = 0$.*

This metatheorem is constructive, and its proof provides a mechanism for extracting computations from classical proofs. But these programs are not (at least, at first blush) naturally related to the classical proofs from which they are obtained. To make the last step, we employ Griffin's [Gri90] result that

- continuation-passing-style translations induce double-negation translations on types of programs in typed λ-calculus,
- the rule of double-negation elimination is a consistent typing for $\mathcal{C}$ (Felleisen's "control" operator [FF86]).

Using these two results, we showed that augmenting the standard program-extraction techniques for HA with $\mathcal{C}$ for the rule of double-negation elimination was sound for PA, and that Friedman's metatheorem was indeed a proof of this fact.

That is, we showed that Friedman's metatheorem could be viewed as either

- a method of translating a classical proof of a Σ_1^0 sentence to a constructive proof, and thence extracting a computation, or
- a method of showing that this augmented extraction method is sound, by CPS-translating $\lambda+\mathcal{C}$ programs, and double-negation/A-translating their typing proofs, observing that the typing is maintained by the translation, and that the original and translated programs have the same values in their respective operational semantics.

Thus we may simply extract programs from (Σ_1^0) classical proofs directly and invoke the metatheorem to show that these proofs are correct. So, for instance, we can show [Mur90] that the call-by-name $\lambda + \mathcal{C}$ calculus, a confluent control language, has a classical type system; the translation used to justify this is a modified Kolmogorov translation. Likewise, a call-by-value $\lambda + \mathcal{C}$ calculus has a classical type system, justified by a modified Kuroda translation.

Given these results, two questions come to mind:

1. These results hold for Σ_1^0 (and, considering free variables, Π_2^0) sentences. To what extent do they hold for formulas higher-up in the arithmetic hierarchy? That is, what is the computational content of, say, the "minimum" proof given earlier?
2. Does this extraction method yield the entire computational content of classical proofs?

The first question bears answering, but is beyond the scope of this paper [Mur91a]. Let it suffice to say that in fact all classical proofs contain computations, but only some of them (and all proofs of Π_2^0 sentences) compute evidence for the propositions they purport to prove. The second question is the subject of this paper, and we begin by examining Felleisen's $\lambda+\mathcal{C}$ calculi in detail.

4 A Programming Language for Classical Proofs

In this section we describe Felleisen's $\lambda + \mathcal{C}$ calculi, and motivate our view that they serve as a framework for viewing classical proofs (particularly as presented in the natural-deduction style) as computations. One can arrive at $\lambda + \mathcal{C}$ by considering a λ-calculus with a deterministic evaluator, and adding two new term-forming operators, $\mathcal{C}$ and $\mathcal{A}$, with evaluation rules. Generally speaking, the choice of a deterministic evaluator is equivalent to choosing an algorithm for separating a program phrase P into two parts - a redex R, and an *evaluation context* (a term with a single "hole" in it), $E[]$, such that $P = E[R]$. Given such an algorithm, we can write the rule of β-reduction for CBN-evaluation as

$$E[(\lambda x.b)N] \quad \succ_1 \quad E[b[N/x]], \quad (\beta)$$

where the right side tells us what to rewrite the entire program, $E[(\lambda x.b)N]$, to. In this notation, the rules for the "control" and "abort" operators are:

$$\begin{array}{lll} E[\mathcal{C}M] & \succ_1 & M(\lambda x.\mathcal{A}E[x]), \quad (\succ_{\mathcal{C}}) \\ E[\mathcal{A}M] & \succ_1 & M, \qquad\qquad\qquad (\succ_{\mathcal{A}}) \end{array}$$

It is instructive to note that $\mathcal{A}$ could be defined as

$$\mathcal{A}M \equiv \mathcal{C}\lambda\_.M,$$

and that call/cc [CR86, BD91] can be defined as

$$\mathcal{K}M \equiv \mathcal{C}\lambda k.k(Mk).$$

The problem with this presentation is that it presupposes a deterministic evaluator, and precludes local equational reasoning about expressions with control operators in them. Felleisen hence "factored" this rule into several rules, most of which could be applied locally, like the β-rule, and some of which could only be applied to complete program expressions. Then, by considering various combinations of these rules, he reconstructed various evaluation strategies, in such a way that the global deterministic evaluator arises as a way of picking standard reduction sequences for an already-confluent reduction relation, in the style of [CF58, Plo75]. In the following rules, we will use $\rightarrow_1$ to denote one-step local *reduction* rules, and $\rhd_1$ to denote one-step global *computation* rules (i.e. only applicable to entire programs). (One version of) Felleisen's rule set is:

$$\begin{array}{rcll} (\lambda x.b)N & \rightarrow_1 & b[N/x] & \beta \\ (\mathcal{C}M)N & \rightarrow_1 & \mathcal{C}\lambda k.M(\lambda f.k(fN)) & \mathcal{C}_L \\ M(\mathcal{C}N) & \rightarrow_1 & \mathcal{C}\lambda k.N(\lambda a.k(Ma)) & \mathcal{C}_R \\ (\mathcal{A}M)N & \rightarrow_1 & \mathcal{A}M & \mathcal{A}_L \\ M(\mathcal{A}N) & \rightarrow_1 & \mathcal{A}N & \mathcal{A}_R \\ \mathcal{C}\lambda g.\mathcal{C}\lambda k.M & \rightarrow_1 & \mathcal{C}\lambda g.M[\lambda x.\mathcal{C}\lambda\_.x/k] & \mathcal{C}\mathcal{C} \\ \mathcal{A}\mathcal{C}\lambda k.M & \rightarrow_1 & \mathcal{A}M[\lambda x.\mathcal{C}\lambda\_.x/k] & \mathcal{A}\mathcal{C} \\ [\mathcal{C}M] & \rhd_1 & M(\lambda x.\mathcal{A}x) & \mathcal{C}_{\rhd} \\ [\mathcal{A}M] & \rhd_1 & M & \mathcal{A}_{\rhd} \end{array}$$

The rules also have versions with restrictions:

$$\begin{array}{rclll} (\lambda x.b)N & \rightarrow_1 & b[N/x] & N \text{ a value} & \beta_v \\ (\mathcal{C}M)N & \rightarrow_1 & \mathcal{C}\lambda k.M(\lambda f.k(fN)) & N \text{ a value} & \mathcal{C}_{Lv} \\ M(\mathcal{C}N) & \rightarrow_1 & \mathcal{C}\lambda k.N(\lambda a.k(Ma)) & M \text{ a value} & \mathcal{C}_{Rv} \\ (\mathcal{A}M)N & \rightarrow_1 & \mathcal{A}M & N \text{ a value} & \mathcal{A}_{Lv} \\ M(\mathcal{A}N) & \rightarrow_1 & \mathcal{A}N & M \text{ a value} & \mathcal{A}_{Rv} \end{array}$$

Then, we can recover, for instance, CBV $\lambda + \mathcal{C}$ (where application nodes are evaluated left-to-right), by using $\beta_v + \mathcal{C}_L + \mathcal{C}_{Rv} + \mathcal{A}_L + \mathcal{A}_{Rv} + \mathcal{C}_{\rhd} + \mathcal{A}_{\rhd}$. In a like manner, we can obtain

CBV (right-to-left) $\lambda + \mathcal{C}$ with $\beta_v + \mathcal{C}_{Lv} + \mathcal{C}_R + \mathcal{A}_{Lv} + \mathcal{A}_R + \mathcal{C}_\triangleright + \mathcal{A}_\triangleright$, and CBN $\lambda + \mathcal{C}$ with $\beta + \mathcal{C}_L + \mathcal{A}_L + \mathcal{C}_\triangleright + \mathcal{A}_\triangleright$. As we mentioned earlier, we can prove that each of these confluent subsystems has a classical typing for Σ^0_1-typed programs. The only rules which are problematic are $\mathcal{C}_\triangleright$ and $\mathcal{A}_\triangleright$. It turns out that in order to justify subject-reduction (to show that $M : \varphi$ and $M \triangleright_1 M'$ imply $M' : \varphi$), one must "pre-A-translate" [Mur91a] the types of terms, so that instead of working in a classical logic with

$H \vdash \mathcal{C}M : P$
  `BY double negation elim`
  $\vdash M : (P \Rightarrow \bot) \Rightarrow \bot$

we instead work with the rule

$H \vdash_\varphi \mathcal{C}M : P$
  `BY double negation elim`
  $\vdash_\varphi M : (P \Rightarrow \varphi) \Rightarrow \varphi$.

We mention this fact only to point out that everything is not completely simple. But one can show that a pseudo-classical typing (as outlined sketchily above) suffices, and that for complete evaluation of Σ^0_1 programs, as opposed to subject reduction, a classical typing is sufficient.

An important fact to remember about our classical type system is that the rule of $\bot$-elimination (which in the pre-A-translated system becomes φ-elimination),

$\vdash M : T$
  `BY ⊥-elim`
  $\vdash M : \bot$.

does not hold. Rather, we "code" $\bot$-elimination via $\neg\neg$-elim:

$\vdash \mathcal{C}(\lambda\_\_.M) : T$
  `BY ¬¬-elim`
  $\vdash \lambda\_\_.M : \neg\neg(T)$
    `BY intro`
    $h : \neg(T) \vdash M : \bot$

Hence our classical system is really obtained by adding $\neg\neg$-elimination to a minimal logic, rather than an intuitionistic logic.

By applying the techniques lifted from the subsystems, to the entire calculus, one gets:

1. a subject-reduction theorem, saying that
   (a) $\vdash_{LK} M : \varphi$ implies $\vdash_{LK(\varphi)} M : \varphi$
   (b) ($\vdash_{LK(\varphi)} M : \varphi$ *and either* $M \rightarrow_1 M'$ *or* $M \triangleright_1 M'$) imply $\vdash_{LK(\varphi)} M' : \varphi$.
2. An evaluation type-soundness theorem, saying that, for $\varphi \in \Sigma^0_1$, $\vdash_{LK} M : \varphi$ and $M \succ_1 V$, where V is a value, imply $\vdash_{LK} V : \varphi$.

However, the entire calculus is not confluent. In fact, the "two primes" example is a good case where the entire calculus gives two type-correct normal forms. If one assigns programs to proofs as suggested herein, one finds that the program assigned to that theorem has two type-correct normal forms: $\langle 5, axiom\rangle$ and $\langle 9, axiom\rangle$. By picking a particular translation from PA to HA, we in effect pick a particular subsystem of $\lambda + \mathcal{C}$. We might conjecture, based on the (to be seen) experience with [BB91]'s system and the Kuroda and Kolmogorov translations, which are all interpretable into Felleisen's system, that Felleisen's system captured the entire computational content of classical proofs.

This conjecture turns out to be false. As we shall see, Girard's new translation from classical to constructive logic induces a new reduction rule upon classical proofs. However, all is not lost. This new rule allows one to perform *evaluation*, and not simply reduction, under a λ-abstraction. Thus one might conjecture that the entire computational content of classical proofs, where *evaluation* under lambda-abstractions is not permitted, is captured by Felleisen's system. This conjecture, though, is not provable, and space does not permit even examples.

Let us remark that the stripped-down $\lambda + \mathcal{C}$ which we have presented so far is not sufficient to stand as an algorithmic account of proofs in arithmetic. It lacks integers, pairing, disjoint union, and the various introduction and elimination rules on these. Thus there are more terms to add, as well as corresponding computation rules. For the most part, we do not need these to convey the intuitive content of this work. However, based on the complete presentation in [Mur90], we believe that the work herein presented extends without effort to the complete programming language of arithmetic.

4.1 New Classical Reductions

In the computer science literature (that we have knowledge of), there are three known classical reductions. Two of these are present in the work of [BB91], and we have never found the latter ($\mathcal{C}_R$) in any *mathematical* literature on classical proofs, but rather only in the continuation/programming-languages community. Hence, we present it here, as an example of the "open-ended-ness" of reduction systems for classical proofs.

Consider the reduction $\mathcal{C}_R$, which converts a term $M(\mathcal{C}N)$ to a term $\mathcal{C}\lambda k.N(\lambda a.k(Ma))$. This reduction, when expressed as one on proofs, converts an instance of $\neg\neg$-elimination on a type A, which is an immediate argument of an $\Rightarrow$-elimination, where the other argument, M, has type $A \Rightarrow B$, to an instance of $\neg\neg$-elimination on a type B. Compare this with the rule $\mathcal{C}_L$, from $(\mathcal{C}M)N$ to $\mathcal{C}\lambda k.M(\lambda f.k(fN))$. This reduction can be seen as converting an instance of $\neg\neg$-elimination on $A \Rightarrow B$ to one on B. Clearly, they are symmetric, in the sense that each is just as reasonable on a program phrase. But only the latter has actually been discovered in the mathematics literature [Pra65], whereas the former was found by Felleisen recently [FFED86].

One cannot be certain that other valid reduction rules will not be discovered with the passage of time. In fact, as we mentioned earlier, and show in section 6, Girard's translation induces a new reduction rule, not found in Felleisen's system, which allows us to evaluate under certain λ-abstractions. Nevertheless, we feel that the reduction system of $\lambda + \mathcal{C}$ is powerful enough to stand as a viable account of the computational content of classical proofs. While it is true that other sensible reduction rules for classical proofs will be found, we feel that any reduction system will in the end have to encode Felleisen's calculus, i.e. be an extension of it.

5 Berardi & Barbanera's Method

[BB91]'s interpretation of classical proofs has two parts - a new and original semantics, for which we refer the reader to their paper [BB91], and a computation system that is a modification of Felleisen's CBN $\lambda + \mathcal{C}$. Their computation system consists of several stages. We lay out the system and then convert each stage to Felleisen's calculus, showing how each of [BB91]'s reductions can be obtained by one or more of Felleisen's rules.

1. A proof of $\exists x.\ f(x) = 0$ is represented as a proof of $\neg(\forall x.\ \neg(f(x) = 0))$. They apply this proof to a proof of $\forall x.\ \neg(f(x) = 0)$, yielding a proof of $\perp$. We use the term $\lambda x.\lambda p.\mathcal{A}\langle x, p\rangle$ for the proof of $\forall x.\ \neg(f(x) = 0)$. This typing is justified by the "pre-A-translation" [Mur91a], mentioned previously. Finally, we wrap a $\mathcal{A}$ around the entire term, to yield $\mathcal{A}(M(\lambda x.\lambda p.\mathcal{A}\langle x, p\rangle))$. This last step is not strictly necessary, but it allows us to leave off the $\mathcal{A}_{\triangleright}$ and $\mathcal{C}_{\triangleright}$ reductions, simplifying the presentation.

2. First computation phase: [BB91] employ two kinds of reductions which, taken together, are confluent. They use β-reduction (and all the other standard reductions of call-by-name HA) viewed as a programming language. Their other reductions correspond to local $\mathcal{C}$-rules, and there are two of real interest:

   $\mathcal{C}_{\Rightarrow}$: Convert

   ```
   ⊢ A ⇒ B BY ¬¬-elim      to   ⊢ A ⇒ B BY intro n
     ⊢ ¬¬(A ⇒ B) ...               n:A ⊢ B BY ¬¬-elim THEN intro k
                                     k:¬(B) ⊢ ⊥ BY cut g:¬¬(A ⇒ B)
                                       ⊢ ¬¬(A ⇒ B) ...
                                       g:¬¬(A ⇒ B) ⊢ ⊥ BY elim g
                                         ⊢ ¬(A ⇒ B) BY intro f
                                           f:A ⇒ B ⊢ ⊥ BY elim k
                                             ⊢ B by elim f
                                               ⊢ A by hypothesis n
   ```

   The corresponding reduction on program expressions is

   $$\mathcal{C}M \rightarrow_1 \lambda n.\mathcal{C}\lambda k.M(\lambda f.k(fn))$$

   where $\mathcal{C}M : A \Rightarrow B$.

   $\mathcal{C}_{\neg}$: Convert

   ```
   ⊢ ¬(A) BY ¬¬-elim       to   ⊢ A ⇒ ⊥ BY intro a
     ⊢ ¬¬(¬(A)) ...                a:A ⊢ ⊥ BY cut g:¬¬(¬(A))
                                     ⊢ ¬¬(¬(A)) ...
                                     g:¬¬(¬(A)) ⊢ ⊥ BY elim g
                                         ⊢ ¬¬(A) BY intro f
                                           f:¬(A) ⊢ ⊥ BY elim f
                                             ⊢ A by hypothesis a
   ```

   The corresponding reduction on programs is

   $$\mathcal{C}M \rightarrow_1 \lambda a.M(\lambda f.fa)$$

   where $\mathcal{C}M : \neg(A)$.

   $\mathcal{C}_{\Rightarrow}$ resembles $\mathcal{C}_L$, and is equivalent to $\mathcal{C}_L + \beta^{-1}$ (inverse β-reduction). $\mathcal{C}_{\neg}$ looks completely different, but it is a composition of $\mathcal{C}_L$, β, $\mathcal{CC}$, and $\mathcal{AC}$, as we shall show later in this section.

3. The reductions β,$\mathcal{C}_{\Rightarrow}$, and $\mathcal{C}_{\neg}$ give a proof in a certain normal form. The Prawitz trivial reduction [Pra81] comes next, and can best be represented by viewing proofs as typing derivations. It converts a proof

$$\begin{array}{lcl} \Gamma \vdash C[M]{:}A \texttt{ BY } \ldots & \quad\text{to}\quad & \Gamma \vdash M{:}A \texttt{ BY } \ldots \\ \quad \Gamma' \vdash M{:}A \ \ldots & & \end{array}$$

when $\Gamma' \subseteq \Gamma$, and $C[]$ is not the "control" operator, but rather an arbitrary term context (the difference in fonts is used to disambiguate). Prawitz' trivial reduction is problematic, because, for instance, a term $S(S(0))$ reduces to $S(0)$ under this scheme. We cannot recover all of the trivial reductions, but we can recover what we feel to be the "intended" subset of these reductions, as we shall see.

Before showing the representation of [BB91]'s computation system in Felleisen's calculus, we discuss briefly the omissions in the presentation. We omit an analysis of products, disjunctions, existentials, universals, and focus on implication and implication-elimination. Nevertheless, these other rules (pair-analysis, case-split, etc) can be easily handled in the same way. Likewise, we do not discuss the atomic rules of arithmetic, e.g. transitivity of equality. This omission presents several problems in discussing [BB91]'s second-stage computations (trivial reduction), but to include them would lengthen the paper considerably. They are described in gory detail in [Mur90], though the application to [BB91]'s system is *not* presented there.

5.1 Reconstructing Berardi & Barbanera's Reductions : First Stage

As we said before, to mimick [BB91]'s first step, we apply M of type $\neg(\forall x.\ \neg(f(x) = 0))$ to $\lambda x.\lambda p.\mathcal{A}\langle x,\, p\rangle$ of type $\forall x.\ \neg(f(x) = 0)$ to yield a term of type $\bot$. We then "wrap" this term in an "abort" to get $\mathcal{A}(M(\lambda x.\lambda p.\mathcal{A}\langle x,\, p\rangle))$, which has type $\exists x.\ f(x) = 0$ in the "pre-A-translated" pseudo-classical type system we mentioned earlier, but $\bot$ in the classical type system.

[BB91]'s $\mathcal{C}_{\Rightarrow}$ reduction is equivalent to $\mathcal{C}_L + \beta^{-1}$. Hence, the confluence of $\mathcal{C}_L + \beta + \mathcal{CC} + \mathcal{AC}$ (asserted here without proof - it follows trivially from the confluence of $\mathcal{C}_L + \beta$ in [Mur90], and, more correctly, by trivial extension of Felleisen's analysis in [FFED86, FH89] - means that we can replace $\mathcal{C}_{\Rightarrow}$ by $\mathcal{C}_L$ without loss of generality, and, more importantly, without changing the normal forms of terms.

[BB91]'s $\mathcal{C}_{\neg}$ reduction is more problematic. Essentially, the reduction is (where both sides have type $A \Rightarrow \bot$):

$$\mathcal{C}M \to \lambda a.M(\lambda f.fa).$$

Since the entire program is of type $\bot$, if we normalize with respect to $\mathcal{C}_L + \beta$, we arrive at a normal form in which every instance $\mathcal{C}M$ (of type $\bot$) is either "encased" in $\mathcal{C}\lambda k.\bullet$ or in the $\mathcal{A}\bullet$ at the top of the program (a crucial fact here is that $\bot$-elim is "coded" in terms of $\neg\neg$-elim: $M : \bot \vdash \mathcal{C}\lambda\_.M : T$). Hence, the $\mathcal{AC}$ and $\mathcal{CC}$ rules apply. Let us look at an example where $M : \neg\neg\neg(A)$ (so $(\mathcal{C}M)N : \bot$):

$$\begin{array}{rcll} \mathcal{C}\lambda g.((\mathcal{C}M)N) & \to & \mathcal{C}\lambda g.\mathcal{C}\lambda k.M(\lambda f.k(fN)) & \textit{by } \mathcal{C}_L + \beta^{-1} \\ & \to & \mathcal{C}\lambda g.M(\lambda f.(\lambda x.\mathcal{C}\lambda\_.x)(fN)) & \textit{by } \mathcal{CC} \\ & \to & \mathcal{C}\lambda g.M(\lambda f.\mathcal{C}\lambda\_.fN) & \textit{by } \beta. \end{array}$$

At this point, it seems that the final term is not equal to $\mathcal{C}\lambda g.M(\lambda f.fN)$. But the key to seeing that in fact we are done is to note that the term $\lambda f.\mathcal{C}\lambda\_.fN$ cannot undergo further reductions until it is applied to some term $F : A \Rightarrow \perp$. At that point, the term $\mathcal{C}\lambda\_.FN$ has type $\perp$, as does the term $F(N)$. Hence, either $\mathcal{C}\lambda\_.FN$ is at the top of the program, in which case the (extra) enclosing $\mathcal{C}\lambda\_.\bullet$ is meaningless, or it is in an evaluation context. At which point we can show that in fact the term $\mathcal{C}\lambda\_.FN$ is "encased" in either $\mathcal{A}\bullet$ or $\mathcal{C}\lambda\_.\bullet$. In either case, a single $\mathcal{CC}$ (or $\mathcal{AC}$) reduction suffices to finish the job.

The idea here is that the type system requires $\mathcal{C}\lambda\_.FN$ to occur in either a top-level or abortive context. This is tied to the fact that $\mathcal{C}\lambda\_.FN$ has type $\perp$. This is indeed the most problematic reduction in our entire interpretation. This method relies implicitly on the fact that we "code" $\perp$-elimination in terms of $\neg\neg$-elimination. As we remarked in section 4, this "coding" ensures that whenever we employ $\perp$-elimination, we encase the term of type $\perp$ in a $\mathcal{C}\lambda\_.\bullet$ There is yet another method of arriving at this same result, though. If, rather than Felleisen's "control" operator, we use his $\mathcal{F}$ operator [Fel87], we can extract programs in much the same way. However, the rules for $\mathcal{CC}$, $\mathcal{AC}$, $\mathcal{C}_{\triangleright}$, and $\mathcal{A}_{\triangleright}$ all are greatly simplified. The drawback of doing this is that the $\mathcal{F}$ operator is not amenable to CPS-representation. Nevertheless, all the classical typing results do go thru for the $\mathcal{F}$ operator.

If we replace $\mathcal{C}\lambda g.\bullet$ with $\mathcal{A}\bullet$, the same reasoning applies, with $\mathcal{AC}$ instead of $\mathcal{CC}$. The representability of $\mathcal{C}_\neg$ with $\mathcal{C}_L + \beta + \mathcal{CC} + \mathcal{AC}$ means that we can use Felleisen's rules, and discard $\mathcal{C}_\neg$ also.

Hence, rather than using $\beta + \mathcal{C}_\Rightarrow + \mathcal{C}_\neg$, we can use $\beta + \mathcal{C}_L + \mathcal{AC} + \mathcal{CC}$. The confluence of this latter system ensures the confluence of the former, and also that the former is simply a subsystem of the latter. Also, seen as program-reduction systems, it seems clear that the latter system is more natural, since reductions do not depend on typings, but rather only on the form of program phrases. [2]

5.2 Reconstructing Berardi & Barbanera's Reductions : Second Stage

To interpret [BB91]'s second computation stage requires an analysis of their stage-I normal forms. Before doing so, though, let's try to understand why we ought to believe that it is even possible to reconstruct trivial reductions in Felleisen's calculus. Recall that a trivial reduction is one that replaces a proof of an atomic proposition A with a proper subproof of the same proposition, provided that the subproof's assumptions are contained in the assumptions of the proof being replaced. That is,

$$C[M] \to M$$

when $C[M] : A$, $M : A$, and the free variables of M are contained in the free variables of $C[M]$ (which is *not* the case, for instance, when $C[]$ binds variables which are then free in M. Consider now the rule for $\mathcal{A}$:

$$\mathcal{A}E[\mathcal{A}M] \to \mathcal{A}M.$$

This reduction clearly looks like a trivial reduction, since both terms have type $\perp$. Likewise, the rule for $\mathcal{C}$:

$$E[\mathcal{C}\lambda k.M] \to M[\lambda x.\mathcal{A}E[x]/k].$$

Any instances of k in M will be replaced by $\lambda x.\mathcal{A}E[x]$. Moreover, any application of k to a term N will reduce as follows:

$$k(N) \to \mathcal{A}E[N]$$

[2]Of course, this is a purely aesthetic judgement; our criterion is clearly based on the belief that a type-free computation system is prior to notions of typing.

and when this $\mathcal{A}$ "fires," it will erase the enclosing evaluation context. Hence, it will appear as if the following occurred:

$$E[\mathcal{C}\lambda k.C[k(N)]] \rightarrow E[N].$$

Again, N has the same type as $\mathcal{C}\lambda k.C[k(N)]$, and it appears as if a trivial reduction has fired. The problem with this intuition is that it does not properly handle trivial reductions which fire across variable bindings. For instance, when x is not free in M, we might like

$$\lambda x.\mathcal{A}M \rightarrow \mathcal{A}M,$$

which is obviously true. Space does not permit handling all of these details, especially in the case where atomic rules like transitivity of equality are allowed. We will discuss the case where no such atomic rules are allowed, and comment on how one would go about verifying that Felleisen's system handled trivial reductions when such atomic rules are allowed.

[BB91] show that their stage-I normal form contains only closed atomic rules. This means that only falsehood, primitive equalities, and negations of primitive equalities occur in the proof. Let us define the normal forms of [BB91]'s calculus. Before reduction begins, a program has the form $\mathcal{A}(M(\lambda x.\lambda p.\mathcal{A}\langle x,\, p\rangle))$. No reduction can "remove" the topmost "abort" (we disallow $\mathcal{A}_{\triangleright}$, so the stage-I normal forms have the form $\mathcal{A}V$, where V contains no stage-I redices, and has type $\perp$. Also, since every type in the proof is atomic (or a negation of an atomic type) and closed, we can immediately conclude that no terms of the form $\lambda x.\lambda p.\mathcal{A}\langle x,\, p\rangle$ exist in V - that each has been applied to some n. The fact that V has type $\perp$ means that no instances of terms $\lambda p.\mathcal{A}\langle x,\, p\rangle$ can occur, since they must be applied, to yield $\perp$, at which point there would exist β-redices. So we can conclude that V must be a term $\mathcal{A}\langle n,\, M\rangle$, where M is a term of *classical* type $f(n)=0$ (remember that the entire program has type $\exists x.\ f(x)=0$.

In turn, M could either be a proof of $f(n) = 0$, in which case the entire program has stage-I-normalized to a term $\mathcal{A}\mathcal{A}\langle n,\, axiom\rangle$, or M could be a control-operator, e.g. $\mathcal{C}\lambda k.M'$. In this case, $k : \neg(f(n) = 0) \vdash M' : \perp$, and an analysis of the structure of M' tells us that it is either an application of k to a term of classical type $f(n) = 0$, or a term $\mathcal{A}\langle n',\, N\rangle$, where $N : f(n') = 0$. This pattern continues throughout the term. Notice that at the point where we find a term $axiom : f(n) = 0$ for some n, the term ends. That is, the inductive analysis we were pursuing closes off the moment we find a true constructive witness. Thus, there is really only one witness in a stage-I-normal form of a proof of $\exists x.\ f(x) = 0$, *provided that no instances of atomic equality reasoning remain in that proof*. This proviso is crucial, as we shall see in the remainder of this section. Presence of transitivity rules, for instance, can allow more than one witness in a proof-tree.

It is illuminating to see the form of these stage-I-normal proofs. The terms are described by a grammar:

$$\begin{array}{rcll} FT(S) & \equiv & \mathcal{A}\langle n,\, EQT_{\langle f(n),0\rangle}(S)\rangle & \\ & | & k(EQT_{\langle n,\, m\rangle}(S)) & (\langle n,\, m\rangle \in S) \\ EQT_{\langle n,\, m\rangle}(S) & \equiv & axiom & (n = m) \\ & | & \mathcal{C}\lambda k : \neg(n = m).FT(\{\langle n,\, m\rangle\} \cup S). & \end{array}$$

Informally, $FT(S)$ describes the form of a term of classical type $\perp$, where S is a set of pairs of natural numbers $\langle n,\, m\rangle$, for each of which there is available, in the lexical environment, a variable of type $\neg(n = m)$. Likewise, $EQT_{\langle n,\, m\rangle}(S)$ is a classical proof of $n = m$, where the proof can use assumptions $\neg(n = m)$ for each $\langle n,\, m\rangle \in S$. The entire program is described by $FT(\{\})$. Each value, then, is a linear sequence of $\mathcal{A}\langle n,\, \bullet\rangle$, $k(\bullet)$, and $\mathcal{C}\lambda k.\bullet$ forms, at the bottom of which we find a *axiom*. Recall now the definition of trivial reduction (quoting [Ber91]):

Replace a proof of an atomic A with any proper subproof of the same A (if any!), [providing] that the assumptions of the subproofs are included in the assumptions of the proof.

Since there is only one *real* proof of $f(n) = 0$ for some n in any stage-I-normal form without standard atomic rules for integers, it is easy to show that the proof in fact normalizes via trivial reductions to the same value in Felleisen's calculus (the subject-reduction property is used here). To get a bit more illuminating proof, we must analyze the structure of the reduction more closely. We can represent the proof by a list of combinators - one for each kind of term in the grammar above. Define

$$\begin{array}{rcl} TOP_n(\bullet) & \equiv & \mathcal{A}\langle n, \bullet\rangle \\ MARK_{\langle f(n),0\rangle,k}(\bullet) & \equiv & \mathcal{C}\lambda k.\bullet \\ THROW_{\langle f(n),0\rangle,k}(\bullet) & \equiv & k(\bullet) \\ AXIOM_{\langle f(n),0\rangle} & \equiv & axiom. \end{array}$$

The intuition here is that we can now represent terms in the grammar above by a string of these combinators. For example,

$$\mathcal{A}\langle n, \mathcal{C}\lambda k.k(axiom)\rangle$$

is represented as

$$TOP_n(MARK_{\langle f(n),0\rangle,k}(THROW_{\langle f(n),0\rangle,k}(AXIOM_{\langle f(n),0\rangle}))).$$

The grammar is now written:

$$\begin{array}{rcll} FT(S) & \equiv & TOP_n(EQT_n(S)) & \\ & | & THROW_{\langle f(n),0\rangle,k}(EQT_n(S)) & (n \in S) \\ EQT_{\langle n,m\rangle}(S) & \equiv & AXIOM_{\langle n,m\rangle} & (n = m) \\ & | & MARK_{\langle n,m\rangle,k}(FT(\{\langle n, m\rangle\} \cup S)), & \end{array}$$

and we can clearly see the linear nature of terms. Felleisen's reductions can then be stated:

$$\begin{array}{rcl} TOP_n(MARK_{\langle f(n),0\rangle,k}(M)) & \to & TOP_n(M[TOP_n/THROW_{\langle f(n),0\rangle,k}]) \\ TOP_n(TOP_m(M)) & \to & TOP_m(M). \end{array}$$

It is a nontrivial, but straightforward, argument to show that we end up with a term $TOP_n(AXIOM_{\langle f(n),0\rangle})$, where $AXIOM_{\langle f(n),0\rangle}$ was present in the original proof.

To extend this presentation to proofs with the atomic rules for integer equality, successor, etc, we need to add the term extractions for these to our language. We will not handle these here, but only conjecture that our analysis extends (with some amount of difficulty) to that case. The modification we must make to the above structure is to add a new injection for EQT:

$$EQT_{\langle n,m\rangle}(S) \equiv TRANS_{n,l,m}(EQT\langle n, l\rangle(S), EQT\langle l, m\rangle(S))$$

where the intuition is that $TRANS_{n,l,m}$ has two subterms - one of which has classical type $n = l$, and the other $l = m$. Both subterms can use the same free variables (S) which are available for the $TRANS$ term. The difficulty then comes in showing that any solution lying in this tree can be, by some sequence of reductions, moved to the root of the program, i.e. become the value of the program under some reduction sequence. The property that makes this easy to prove is the representation

$$TRANS_{n,l,m}(M, N) \equiv (\lambda^v\ u'.\lambda^v\ v'.axiom)(M)(N)$$

where λ^v is the call-by-value lambda-abstraction [Mur90, Mur91b]. This allows embedded continuations in *either* M or N to "fire" (by applying different combinations of $\mathcal{C}_L$, $\mathcal{C}_R$, $\mathcal{A}_L$, $\mathcal{A}_R$) and thus for "answers" in M or N to "emerge."

5.3 Differences Between Berardi & Barbanera's and Felleisen's Calculus

There are some huge gaps in our presentation, and also some remaining differences between the "answers" obtained by using Felleisen's calculus, and those obtained using [BB91]'s calculus. First, we have not actually shown that each trivial reduction can be simulated by some sequence of Felleisen's reductions. Rather, we have shown that all the "answers" obtained by [BB91]'s method, in the case where a proof contains no atomic rules such as transitivity, are obtained by applying Felleisen's calculus. Of course, this was made easy by the fact that there was only one answer to be obtained. Also, Prawitz' trivial reduction can produce answers which must be discarded later. For instance, as remarked in [Ber91], though the method of Prawitz' trivial reduction works for systems where atomic propositions have no true computational content (as in the case of equalities), in simply-typed systems this breaks down, since a classical proof $S(S(0))$ of N would Prawitz-reduce to $S(0)$ and also to 0. We might want, though, for these latter two to be disallowed, since the proof above $S(S(0))$ clearly does not specify 0 as an answer.

One way to do this is to use the information in the classical proof about which terms are actually supposed to be trivially reduced. This information is contained in the pattern of applications of continuations (the variables k of type $\neg(f(n) = 0)$ in the discussion above), and it is exactly this information that Felleisen's reduction uses to convert terms to a normal-form which is free of control operators.

6 Girard's Translation

In this section, we begin the process of interpreting Girard's translation from NK to NJ, by describing the translation on the simply-typed $\lambda + \mathcal{C}$, with primitive constants, but without function constants. So, for instance, the simply-typed $\lambda + \mathcal{C}$ built upon the integers, but without primitive operations on the integers, is an example.

Girard's translation defines an equivalence relation on NK proofs by pulling back the equivalence on NJ proofs thru the translation; that is, two NK proofs M and M' are equal in NK exactly when their translations, $\underline{M}$ and $\underline{M'}$, are equal in NJ.

We feel we should point out that our presentation is geared towards translating, top-down, program terms extracted from classical proofs. Thus, we choose to use a natural-deduction version of classical implicational logic, and not a sequent version with "cut." Also, rather than minimizing the extra λ-abstractions, we chose to allow them to appear. The difference between our presentation and Girard's in [Gir91] lies in our choice of a natural-deduction presentation of proofs/programs, and in our using a version of Girard's translation that stands in relation to his in the same manner that standard CPS-translation stands to Plotkin's colon-translation [Plo75]. That is, Girard's translation is an optimized version of ours in the same manner that CBV (respectively CBN) colon-translation is an optimized version of CBV (CBN) CPS-translation.

6.1 The Translation Itself

Upon examining Girard's translation, we find (as will become apparent) that it is an "ultra-by-value" translation. The translation is defined via a notion of "positive" and "negative" formulas. Intuitively, a "positive" formula is a data-value. Falsehood ($\perp$) is positive, as is Truth ($\boldsymbol{V}$). This is consistent with our earlier idea of "pre-A-translation", wherein we replaced every instance of $\perp$ with the type of the entire program. In this case, that type will always be a data-type, since the analogue of Σ_1^0 in a simply-typed system is a data-type. Atomic propositions, e.g. A, that stand for primitive data-types, e.g. Int, are also positive. Negations of atomic formulas are negative.

Intuitively compound types are positive if they are guaranteed to contain data (somehow). Intuitively, again, a negative formula is one that has values which may contain continuations. For example, $A \wedge B$ is positive because A,B are positive. But so is $A \wedge \neg(B)$, since A is positive. $\neg(A) \wedge \neg(B)$ is negative, since this is a type of pairs of continuations - there is no possibility of there being data in such a type. Likewise, $A \vee B$ is positive, but $\neg(A) \vee B$, $\neg(A) \vee \neg(B)$ are negative - a value of either type might be a pure continuation. Paradoxically (at first), the type $\neg\neg(A)$ is positive, since the type is really a type of values. To see this, consider that this type is one of functions which accept A-continuations, and "finish" the computation. Hence, if they use the A-continuation at all, they must apply it to a value. This value, then, is what is computed in the process of computing a value of type $\neg\neg(A)$. And, in fact, the translation of $\neg\neg(A)$ is equal to the translation of A.

Negating a formula flips polarity, and implication can be understood in terms of its definition in terms of disjunction and negation. The definition of polarity, lifted from [Gir91] and restricted to the implicational part, is shown below:

A	B	$A \Rightarrow B$	$\neg(A)$
$+$	$+$	$-$	$-$
$-$	$+$	$+$	$+$
$+$	$-$	$-$	
$-$	$-$	$-$	

Girard's translation on types, then, is by cases on polarity of subterms of compound formulas (we write $\overline{\bullet}$ for the translation):

$\overline{A}$	$\overline{B}$	$\overline{A \Rightarrow B}$	$\overline{\neg(A)}$
P	Q	$\neg(P \wedge \neg(Q))$	$\neg(P)$
$\neg(P)$	Q	$P \vee Q$	P
P	$\neg(Q)$	$\neg(P \wedge Q)$	
$\neg(P)$	$\neg(Q)$	$\neg(\neg(P) \wedge Q)$	

The beautiful thing about this translation is that

$$\overline{A \wedge B \Rightarrow C} \equiv \neg((A \wedge B) \wedge \neg(C))$$

and

$$\overline{A \Rightarrow B \Rightarrow C} \equiv \neg(A \wedge (B \wedge \neg(C))).$$

So the isomorphism between $A \wedge B \Rightarrow C$ and $A \Rightarrow B \Rightarrow C$ in the constructive *target* system (e.g. LJ) implies the existence of the isomorphism for the classical source system. There are many other beautiful isomorphisms which hold for this translation also [Gir91]. Another nice thing about this translation is that the translation of a negative formula is always outermost negated, while that of a positive formula never is.

If we had chosen the full calculus, we would have been faced with nondeterminism in the translation on programs. This choice is akin to the choice we must make when CPS-translating $M(N)$, where we must choose between

$$\begin{gathered}\lambda k.\underline{M}(\lambda m.m(N)k)\\ \lambda k.\underline{M}(\lambda m.\underline{N}(\lambda n.m(n)k))\\ \lambda k.\underline{N}(\lambda n.\underline{M}(\lambda m.m(n)k)).\end{gathered}$$

By restricting to the implicational fragment, the nondeterminism disappears. Thus, classical proofs in the implicational fragment can only have one answer. Of course, the proof $\mathcal{C}\lambda k(\mathcal{A}(k1))(\mathcal{A}(k2))$ is an example of one in which there are two answers. Girard's translation would choose which answer was produced by examining the types of terms in the program. So if $\mathcal{A}(k1)$ had a negative type, the entire program would return 2. Otherwise, it would return 1.

Girard's translation on terms of the implicational fragment is defined with reference to the types of subterms. We will handle each case (variables, λ-abstractions, and applications) separately. For each, we must specify the types of all terms and immediate subterms, and their polarities. At various times in the discussion to follow, we will use the terminology A^+ (A^-), to mean that A is a positive (resp. negative) type.

$x{:}A$ **- a variable:**

A	$\underline{x}$
$+$	$\lambda k.k(x)$
$-$	x

$\lambda x.M{:}(A \Rightarrow \bot)$**:**

$$\underline{\lambda x.M} \equiv \lambda k.(\underline{M}[k/x]\lambda x.x)$$

$M(N)$ **-** $M{:}(A \Rightarrow \bot)$**:**

A	$\underline{M(N)}$
$+$	$\lambda k.\underline{M}(\underline{N})$
$-$	$\lambda k.\underline{N}(\underline{M})$

$\lambda x.M{:}(A \Rightarrow B)$ **where** $(B \neq \bot)$**:**

A	B	$\underline{\lambda x.M}$
$+$	$+$	$\lambda d.spread(d; x, k'.\underline{M}k')$
$+$	$-$	$\lambda d.spread(d; x, k'.\underline{M}k')$
$-$	$-$	$\lambda d.spread(d; x, k'.\underline{M}k')$
$-$	$+$	$\lambda k.(\underline{M}[k \circ inl/x])(k \circ inr)$

$M(N)$ - M:$(A \Rightarrow B)$, $B \neq \bot$:

A	B	$\underline{M(N)}$
$+$	$+$	$\lambda k.\underline{N}(\lambda n.\underline{M}\langle n,\, k\rangle)$
$+$	$-$	$\lambda k.\underline{N}(\lambda n.\underline{M}\langle n,\, k\rangle)$
$-$	$-$	$\lambda k.\underline{M}\langle \underline{N},\, k\rangle$
$-$	$+$	$\lambda k.\underline{M}(\lambda d.decide(d; p.\underline{N}p; q.k(q)))$

$\mathcal{C}M$:

$$\underline{\mathcal{C}M} \equiv \underline{M}$$

$\mathcal{A}M$:

$$\underline{\mathcal{A}M} \equiv \lambda k.(\underline{M}\lambda x.x)$$

There are several ways to arrive at this translation. One can begin with Girard's translation as specified in [Gir91], and attempt to infer where certain complex terms should be simplified by reverse-β-reduction. This resembles being given a colon-translation, and being asked to present the associated CPS-translation in standard form. Alternatively, one can use Girard's translation as a blueprint, derive a set of domains and a denotational semantics for classical proofs, in a manner identical to that done for the lambda-calculus in any standard text on denotational semantics, and defunctionalize this semantics to arrive at the above translation.

We will not attempt to verify reduction rules from Felleisen's calculus, since this is a time-and-space-consuming task. However, we observe that it is trivial to see that if $\mathcal{A}M$ is of a positive type, e.g. Int, then $(\mathcal{A}M)N \rightarrow_1 \mathcal{A}M$, and $N(\mathcal{A}M) \rightarrow_1 \mathcal{A}M$. Basically, focusing on the translations of λ-abstraction and application when $B \neq \bot$, and ignoring the case where the polarities are $(-,+)$, the translation appears to be a standard call-by-value CPS-translation.

Recall from the call-by-value CPS-translation that there are two different ways to CPS-translate application:

$$\begin{array}{rcl} \underline{M(N)} & \equiv & \lambda k.\underline{M}(\lambda m.\underline{N}(\lambda n.m(n)k)) \\ & or & \lambda k.\underline{N}(\lambda n.\underline{M}(\lambda m.m(n)k)) \end{array}$$

In a manner which can be made precise, these correspond to evaluating the subterms of an application node in, respectively, left-to-right and right-to-left order, before contracting the β-redex. Going back to the translation of application, we see that when A is positive, $N : A$ is evaluated first.

There is more going on, though, because, if we look at the rule for λ-abstraction, polarity $(-,+)$, we see an amazing thing. In most standard CPS-translations, the translation of $\lambda x.M$ is something like $\lambda k.k(\lambda x.\underline{M})$. This translation is consistent with the assumption that $\lambda x.\underline{M}$ is a value. However, if $x : A$ is negative, and M, under that typing assumption, is positive, the translation does not treat the λ-abstraction as being a value. Indeed, in this case, the type $A^- \Rightarrow B^+$ is positive! Thus, the translation computes a positive data-value from this expression.

We can gain an understanding of why the expression $\lambda x.M$ might compute a data-value, if we think about symbolically evaluating it after all the free variables in M (except x) have been given types. Since x is of negative type, it is really a continuation. Let us assume (for now) that x has type $A \equiv D \Rightarrow \bot$, where D is a data-type (e.g. positive). Now, in symbolically evaluating M, we will either apply x to a value, or end up computing a data-value, V, of type B. This value might contain embedded instances of x, but it will also contain some real data, since B is a positive type. If x is applied, say, to a value N, then we can replace $\lambda x.M$ with $\lambda x.x(N)$. If x is not applied, we may replace $\lambda x.M$ with $\lambda x.V$. In either case, N (respectively V) is a data-value. So when a consumer of $\lambda x.x(N)$ (resp. $\lambda x.V$) applies this term to an argument, no internal computation need be done.

To make this precise, we can "guess" a new reduction for classical proofs. We can then use the above machinery, and verify that the reduction is in fact correct. Suppose

$$\begin{array}{rcl} L & : & (A^- \Rightarrow B^+) \Rightarrow C^+ \\ M & : & B^+ \ (\mathit{when}\ x : A^-). \end{array}$$

Then

$$L(\lambda x.M) : C^+$$

and

$$\mathcal{C}\lambda c.((\lambda b.c(L(\lambda\_.b)))(M[\mathcal{C}\lambda k.c(L(\lambda a.\mathcal{A}(ka)))/x])) : C^+$$

have the same type, and the same CPS-translation, by the rules given above. A proof of this fact involves lengthy, tedious computation, but that is all. This equation can be added to Felleisen's calculus as a new reduction rule, but at the cost of sacrificing the treatment of λ-abstraction as a value.

6.2 An Example

The previous discussion pointed out two rules which are preserved ($\mathcal{A}_L$ and $\mathcal{A}_R$), and a new rule which must be added to any reduction system for classical proofs which is sound with respect to Girard's translation. We do not yet have a complete characterization of Girard's translation in terms of the reduction rules it induces on classical proofs. However, we can evaluate classical proofs produced by the translation, and observe the results. In this section, we give one example.

$$\mathcal{C}\lambda k.(\lambda p.k(1))(\mathcal{C}\lambda g.k(2)).$$

This expression can be more succinctly (though not completely correctly) written as

$$(\lambda p.1)(\mathcal{A}2).$$

Suppose that p has type N. It should be clear that the translation will produce a program which returns only 2, and not 1. In fact, the program computed is

$$\lambda k.[(\lambda x.x)((\lambda x.x)(k2))]$$

or, after beta-reduction,

$$\lambda k.k2.$$

7 Issues and Conclusions

The programming language of classical logic is inherently multi-valued. The process of double-negation translation, as in Griffin, Murthy, or Girard, chooses a particular value, by fixing an evaluation order. Equivalently, it fixes a set of reductions out of the entire set discovered by Felleisen, and a precedence on when they may be used. If there are different double-negation translations available, then there may be many different values chosen, but it seems that no set of double-negation translations could ferret out all the answers in a classical proof. This conjecture, though, does not yet have a formal proof.

By starting with Felleisen's full calculus, in its nondeterministic, nonconfluent entirety, we admit the possibility of evaluation orders which are not amenable to representation via double-negation translation. Hence, one might conjecture that with such a calculus, every answer in a classical proof is computable by some evaluation order. And, in fact, we can represent the classical computation system of Berardi and Barbanera as an evaluation system that comes in two phases - the first is confluent, and converts a term to a $\beta + \mathcal{C}_L + \mathcal{CC} + \mathcal{AC}$-normal form, and the latter employs more rules of the calculus in a nonconfluent manner to extract all the answers left in the stage-I-normal proof. The first stage, however, is sound with respect to call-by-name CPS-translation, and hence will discard some of the answers that arise thru call-by-value evaluation.

Bolstered by this success, we might look next at Girard's translation, and expect that, again, we could represent his evaluation system by a set of Felleisen's reduction rules. Alas, this turns out to be impossible. Girard's translation evaluates under λ-abstractions (in order to preserve the isomorphism between, e.g. $\neg\neg(Int)$ and Int), and this evaluation method is inherently unrepresentable in Felleisen's calculus as it stands. Thus, we must add at least one, and perhaps more, rules to Felleisen's calculus in order to represent Girard's translation.

However, the more general theme of this investigation stands intact, for this new reduction can be integrated in a meaningful manner with the other reductions - the new reduction allows us to evaluate under a λ-abstraction of positive type (which, in this context, means that it accepts a continuation as argument, and produces a data-value as result) to a value, either to be fed to its argument, or to the current continuation.

References

[BB91] F. Barbanera and S. Berardi. Program extraction for classical logic, 1991. BRA-LF 1991 paper.

[BD91] D. MacQueen B. Duba, R. Harper. Typing first-class continuations in ml. pages 163–173, 1991.

[Ber91] S. Berardi. Personal communication, 1991.

[Bis70] Errett Bishop. Mathematics as a numerical language. In et al. J. Myhill, editor, *Intuitionism and Proof Theory*, pages 53–71. North-Holland, 1970.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[chetCon71] R. L. Constable. Constructive mathematics and automatic program writers. In *Proc. IFP Congr.*, pages 229–33, Ljubljana, 1971.

[Con86] Robert L. Constable, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CR86] W. Clinger and J. Rees. The *revised*^{3} report on the algorithmic language scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

[Fel87] Matthias 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.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–141. North-Holland, 1986.

[FFED86] 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.

[FH89] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice, Houston, 1989.

[Fri78] Harvey 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.

[God65] Gödel, Kurt. On intuitionistic arithmetic and number theory. In Davis, M., editor, *The Undecidable*, pages 75–81. Raven Press, 1965.

[chetGir87] Jean-Yves Girard. *Proof Theory and Logical Complexity, vol. 1.* Bibliopolis, Napoli, 1987.

[Gir91] J.-Y. Girard. A new constructive logic : Classical logic, 1991. Unpublished draft.

[Gri90] Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, 1990.

[How80] W. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 479–490. Academic Press, NY, 1980.

[Kol67] A. N. Kolmogorov. On the principle of the excluded middle. In Jean van Heijenoort, editor, *From Frege to Gödel: A Source Book in Mathematical Logic, 1879–1931*, pages 414–437. Harvard University Press, Cambridge, Massachusetts, 1967.

[ML82] Per 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.

[Mur90] Chet Murthy. *Extracting Constructive Content from Classical Proofs.* PhD thesis, Cornell University, Department of Computer Science, 1990.

[Mur91a] Chet Murthy. Classical proofs as programs : How, when, and why. In *Proceedings of the First Annual Symposium on Constructivity in Computer Science*, 1991.

[Mur91b] Chet Murthy. An evaluation semantics for classical proofs. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*, 1991.

[Plo75] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[Pra65] Dag Prawitz. *Natural Deduction, A Proof-Theoretical Study.* Almqvist & Wiksell, 1965.

[Pra81] Dag Prawitz. Validity and normalizability of proofs in 1st and 2nd order classical and intuitionistic logic. In *Atti del 1st congresso italiano di Logica*, pages 11–36. Bibliopolis, 1981.

[Sza] Szabo, editor. *The Collected Works of Gerhard Gentzen.* North-Holland.

Order-Sorted Polymorphism in Isabelle[‡]

Tobias Nipkow[§]
University of Cambridge
Computer Laboratory
Pembroke Street
Cambridge CB2 3QG
England

June 1991

Abstract

ML-style polymorphism can be generalized from a single-sorted algebra of types to an order-sorted one by adding a partially ordered layer of "kinds", "sorts", or "classes" on top of the types. The resulting type system has been implemented in Isabelle to permit type variables to range over user-definable subsets of all types. Order-sorted polymorphism allows a simple specification of type restrictions in many logical systems. It can also explain Standard ML's equality types and Haskell's type classes. Order-sorted unification is the basis for a simple extension of the Hindley/Milner type inference algorithm.

1 Introduction

This paper describes a recent extension of the generic theorem prover Isabelle. The extension, namely the introduction of "order-sorted polymorphism", was motivated by the desire to obtain simple axiomatizations of typed object-logics. The rest of the introduction is devoted to Isabelle and the motivation for its extension. Section 2 describes the technicalities underlying the new type system and presents a number of object-logic definitions to demonstrate its flexibility. Section 3 shows applications of order-sorted polymorphism to type systems for programming languages, in particular ML and Haskell.

Isabelle is a generic interactive theorem prover developed by Lawrence Paulson [16, 17]. It is generic in the sense that it can be parameterized with the intended object-logic. Its meta-logic is intuitionistic higher-order logic and its inference mechanism is based on higher-order unification. In order to distinguish the old and the new version of Isabelle I will use the qualifiers "90" and "91", respectively.

The presentation of an object-logic in Isabelle-90 consists of

- a signature extension, introducing the new types and constants representing the syntax of formulae, and

‡Isabelle is written in Standard ML. The source files and documentation can be obtained free of charge by contacting the author.

§E-mail: `Tobias.Nipkow@cl.cam.ac.uk`. Research supported by ESPRIT BRA 3245, *Logical Frameworks*.

- new axioms describing the inference rules of the logic.

In the sequel I use an OBJ-inspired [5] syntax which is very close to the format actually used in the implementation. As a tiny example, consider the following definition of propositional logic.

$$\begin{array}{rcl} \mathcal{P}rop & = & \textbf{types}\ form \\ & & \textbf{consts}\ \_\&\_ : form \to form \to form \\ & & \qquad\quad \neg : form \to form \\ & & \qquad\quad [\_] : form \to prop \\ & & \textbf{rules}\ [P] \Longrightarrow [Q] \Longrightarrow [P\&Q] \\ & & \qquad ([\neg P] \Longrightarrow [P]) \Longrightarrow [P] \end{array}$$

The only new type is $form$, the type of formulae. The only connectives are conjunction (&) and negation ($\neg$). In Isabelle-90, constants can be of base type or of arbitrary function type over the base types. Although the absence of a nullary constant, for example $true$, means there are no closed formulae, expressions like $P\&Q$, where P and Q are variables, are legal, provided both P and Q are of type $form$. Variables need not be declared and their type is inferred automatically: writing $P\&Q$ forces P and Q to have type $form$. Finally there is the constant $[\_]$ which maps $form$ to the predefined type $prop$ of propositions. The proposition $[P]$ should be read as "The formula P is provable". This coercion is necessary because object-level inference rules are written as meta-level axioms, i.e. expressions of type $prop$. For example the first rule above is the linear form of the text-book inference

$$\frac{P \quad Q}{P\&Q}$$

using the pre-defined meta-level implication $\Longrightarrow$ of type $prop \to prop \to prop$. Hence it is impossible simply to write $P \Longrightarrow Q \Longrightarrow P\&Q$. In the sequel square brackets are dropped. This improves readability and is in line with actual Isabelle practice, where parser and pretty-printer take care of such matters.

This is an intentionally fragmentary presentation of propositional logic — many rules and connectives are missing. Throughout the paper I adopt the convention of presenting only as much of a logic as is necessary to make certain points.

We can now extend propositional logic to a single sorted first-order logic by adding a new type of terms and a quantifier:

$$\begin{array}{rcl} \mathcal{P}red & = & \mathcal{P}rop\ +\ \textbf{types}\ term \\ & & \qquad\quad \textbf{consts}\ \forall : (term \to form) \to form \\ & & \qquad\quad \textbf{rules}\ \forall(P) \Longrightarrow P(t) \end{array}$$

Because Isabelle is based on higher-order logic, its expressions are simply typed λ-terms. Quantifiers can thus be expressed as functionals. This means that internally the formula $\forall x.P(x)$, where P is of type $term \to form$, is represented as $\forall(\lambda x.P(x))$, which is equivalent to $\forall(P)$. Again, the parser and pretty-printer permit all these different representations. The one inference rule shown is *$\forall$-elimination* or *specialization*.

This definition of predicate logic is fine as far as it goes, but has one major drawback: it is single-sorted, i.e. there is only a single type $term$ of individuals. Any attempt to do many-sorted logic on top of it is as prone to type errors as programming in an untyped language. If we were to extend $\mathcal{P}red$ further by two constants $succ : term \to term$ and $[] : term$ representing the successor function on natural numbers and the empty list, the (in our understanding) meaningless expression $succ([])$ is well-formed and of type $term$. There are several ways to deal with the problem of representing typed object-logics in Isabelle.

The generic representation of typed logics in Isabelle, and in any other such system, is to introduce types and type checking into the object-language. This means there is a new type of object-level types and the inference rules are augmented to ensure that only well-typed theorems can be deduced. Under this approach, the definition of $\mathcal{P}red$ might look as follows:

$$\begin{array}{rcll} \mathcal{P}red & = & \mathcal{P}rop \;+ & \textbf{types}\; term, type \\ & & & \textbf{consts} \quad \forall : type \to (term \to form) \to form \\ & & & \qquad\quad \_ \in \_ : term \to type \to prop \\ & & & \textbf{rules}\; t \in T \Longrightarrow \forall(T,P) \Longrightarrow P(t) \end{array}$$

The universal quantifier has acquired a new argument, namely the type that the bound variable is ranging over. The judgement $\in$ is introduced to form explicit typing assertions at the object-level. The rule of specialization is guarded by the expression $t \in T$ which makes sure that only elements of the right type are substituted for the bound variable.

A further extension with new object-level types, say the natural numbers, might look like this:

$$\begin{array}{rcll} \mathcal{N}at & = & \mathcal{P}red \;+ & \textbf{consts} \quad nat : type \\ & & & \qquad\qquad\quad 0 : term \\ & & & \qquad\quad succ : term \to term \\ & & & \textbf{rules} \;\; 0 \in nat \\ & & & \qquad\quad n \in nat \Longrightarrow succ(n) \in nat \end{array}$$

In the general case this explicit axiomatization of typing rules is the only way to proceed, in particular if the type system is undecidable, for example Intuitionistic Type Theory [9]. On the other hand, it is frustrating to deal with object-logic typing judgements for simple systems like many-sorted predicate logics. Not only does it clutter up the specification of the logic, but it also introduces all these additional type checking obligations during a proof. Although all of these proof obligations can be dealt with automatically by tactics, the incurred overhead is often significant. This is particularly annoying because Isabelle has a built-in type inference system which is quite capable of dealing with many sorted logics, provided the meta-logic type system is used.

The idea of identifying meta-logic and object-logic types is the key to inheriting Isabelle's built-in facilities. Of course it presupposes that the object-logic type system is weak enough. It fails for logics with dependent types or subtypes, neither of which can be modelled by the simple meta-types. In fact, in Isabelle-90 the idea does not work at all, as we shall now see. It is straightforward to declare different meta-types like nat and $list$ and constants $succ : nat \to nat$ and $[] : list$, thus preventing expressions like $succ([])$. However, we now need a quantifier for each of these types, i.e. $\forall_{nat} : (nat \to form) \to form$ and $\forall_{list} : (list \to form) \to form$, and the corresponding rules for each of them. This may just about be feasible for a small number of fixed types but is completely impractical in general.

In order to make this approach work, we need polymorphism, which is exactly what Isabelle-91 and this paper are all about. However, a naïve adoption of ML-style polymorphism is dangerous, as the following example shows:

$$\begin{array}{rcll} \mathcal{P}red & = & \mathcal{P}rop \;+ & \textbf{consts}\; \forall : (\alpha \to form) \to form \\ & & & \textbf{rules}\; \forall x.P(x) \Longrightarrow P(t) \end{array}$$

The universal quantifier has a polymorphic type. The type variable α is implicitly universally quantified at the level of types, as is the usual practice. Type inference determines the following typings of the variables in the specialization rule: $x : \alpha$, $P : \alpha \to form$ and $t : \alpha$. Unfortunately, this specification is too general, at least for first-order logic. The intention is that α ranges only over different types of "individuals", but certainly not over formulae or arbitrary function types. However, there is nothing in this declaration to enforce those constraints. As a consequence, some rather surprising inferences are possible. Using the substitution $\{\alpha \mapsto form, P \mapsto \lambda x.x\}$, specialization degenerates to $\forall x.x \Longrightarrow t$. In a first order-logic, $\forall x.x$ is simply ill-formed. This formula could only arise because of the instantiation of α by $form$. In a higher-order logic, $\forall x.x$ is a perfectly legitimate formula, usually identified with falsity, and the rule expresses *ex falso quodlibet.*

2 Order-Sorted Polymorphism

Having demonstrated the need for tighter control over instantiations of type variables, this section describes a simple extension of the Hindley/Milner type discipline which achieves just that without sacrificing any of the properties that make Hindley/Milner polymorphism attractive. The basic idea is to introduce an additional layer of *sorts* on top of the types, together with a new judgement $\tau : s$, where τ is a type and s a sort. This means in particular that type variables have a sort, thus restricting their instantiations to types of the required sort. Sorts denote subsets of the set of all types. In order to specify hierarchies of such subsets, sorts can be partially ordered, thus leading to an order-sorted algebra of types. It must be stressed that there is no notion of subtypes involved. This is in contrast to, for example, OBJ [5], where the types[1] themselves are partially ordered.

Although this notion of sorts is superficially similar to the notion of "kinds" in the Edinburgh Logical Framework [6], the reader will see that it is qualitatively quite different, not least because of the partial order on sorts. It is much more closely related to the Haskell notion of "classes", and Section 3 shows how to interpret Haskell's type classes in order-sorted polymorphism.

We will now introduce the different features of order-sorted polymorphism by example, going through a series of logic definitions in Isabelle-91, starting with Isabelle's meta-logic. As we are only interested in the type system, none of the inference rules are shown.

$$\begin{array}{rcll}
\mathcal{M}eta & = & \textbf{sorts} & \top \\
 & & \textbf{types} & _ \to _ : (\top, \top)\top \\
 & & & \quad prop : \top \\
 & & \textbf{consts} & \quad \bigwedge : (\alpha_\top \to prop) \to prop \\
 & & & _ \Longrightarrow _ : prop \to prop \to prop \\
 & & & _ == _ : \alpha_\top \to \alpha_\top \to prop
\end{array}$$

Since $\mathcal{M}eta$ is not an extension of anything else, we start from ground zero, i.e. no sorts, types etc. are known. The **sorts** section, not present in Isabelle-90, introduces a single new sort $\top$. The intention is that $\top$ should be the sort of all types. The **types** section is different from its Isabelle-90 counterpart in that it does not just list the types but also gives them an arity. Here there are two types, namely the base type *prop* of sort $\top$ and the function type constructor $\to$ which takes two arguments of sort $\top$ and returns a result of sort $\top$. The notation $t : (s_1, \ldots, s_n)s$ was chosen instead of something like $t : s_1 \to \cdots \to s_n \to s$ in order not to overload $\to$. Note

[1] called *sorts* in that context, making the terminology somewhat confusing.

that with respect to the current type signature, all types built from *prop* and $\to$, i.e. all ground types, are indeed of sort $\top$.

The **consts** section introduces the three basic connectives of the meta-logic: universal quantification, implication, and equality. The type of $\bigwedge$ is very general, i.e. it is possible to quantify over *prop* and over arbitrary function types. Thus we have defined an impredicative meta-logic. A predicative variant could have been defined as follows:

$$
\begin{array}{lcl}
\mathcal{M}eta' & = & \textbf{sorts } \top \\
 & & \qquad i < \top \\
 & & \textbf{types } \_ \to \_ : (\top, \top)\top \\
 & & \qquad prop : \top \\
 & & \textbf{consts} \qquad \bigwedge : (\alpha_i \to prop) \to prop \\
 & & \qquad \_ \Longrightarrow \_ : prop \to prop \to prop \\
 & & \qquad \_ == \_ : \alpha_\top \to \alpha_\top \to prop
\end{array}
$$

Here we introduced a subsort i of "individuals". So far, no type lies in i, but the intention is that types introduced in object-logics should reside there if they are to be quantified over. Meta-level quantification over *prop* or function types is excluded because neither are of sort i. Thus we have arrived at a predicative meta-logic.

Object-logics are defined as extensions of the meta-logic. In order to demonstrate the full flexibility of order-sorted polymorphism we look at definitions of first-, second-, and higher-order logic.

$$
\begin{array}{lcl}
\mathcal{FOL} & = & \mathcal{M}eta + \textbf{sorts } i < \top \\
 & & \qquad \textbf{types } form : \top \\
 & & \qquad \textbf{consts } True : form \\
 & & \qquad\qquad \_ = \_ : \alpha_i \to \alpha_i \to form \\
 & & \qquad\qquad \forall : (\alpha_i \to form) \to form
\end{array}
$$

Quantification over and equality between formulae or function types is ruled out because neither has sort i. For example $True = True$ is not well-typed. The sort i of "individuals" is initially empty but further extensions may change this:

$$
\begin{array}{lcl}
\mathcal{N}at & = & \mathcal{FOL} + \textbf{types } nat : i \\
 & & \qquad \textbf{consts} \qquad 0 : nat \\
 & & \qquad\qquad succ : nat \to nat
\end{array}
$$

The formula $\forall x. x = 0$ is legal, with x having the inferred type *nat* which is of sort i. On the other hand $\forall f. f(0) = 0$ is ill-typed because f must have type $nat \to nat$ which is not of sort i.

A more liberal dose of quantification is granted in second-order logic:

$$
\begin{array}{lcl}
\mathcal{SOL} & = & \mathcal{M}eta + \textbf{sorts } q < \top \\
 & & \qquad\qquad i < q \\
 & & \qquad\qquad f < q \\
 & & \qquad \textbf{types } form : f \\
 & & \qquad\qquad \_ \to \_ : (i, f)f \\
 & & \qquad \textbf{consts } \forall : (\alpha_q \to form) \to form
\end{array}
$$

This type structure is more involved. The sort q denotes all types that one can quantify over. These are all individuals (hence $i < q$) and all n-ary predicates of type $\tau_1 \to \cdots \to \tau_n \to form$, where each τ_j is a type of sort i, such as nat. The latter set of types is called f and there are two declarations which generate types in f: $form$ is in f, corresponding to the case $n = 0$, and if $\tau : i$ and $\sigma : f$, then $\tau \to \sigma$ is also in f. It should be clear that this defines exactly the intended set of predicate types.

$\mathcal{SOL}$ is interesting because it presents a new feature of the type system, namely overloading: the function type constructor has two arities, $(\top, \top)\top$, which is inherited from $\mathcal{M}eta$, and $(i, f)f$. Neither subsumes the other: $form{\to}form : \top$ and $nat{\to}form : f$ require the first and second arity respectively.

Another example of overloading is found in the definition of higher-order logic:

$$\begin{array}{rclll} \mathcal{HOL} & = & \mathcal{M}eta & + & \textbf{sorts } i < \top \\ &&&& \textbf{types } form : i \\ &&&& \qquad\quad \_ \to \_ : (i,i)i \\ &&&& \textbf{consts } \forall : (\alpha_i \to form) \to form \end{array}$$

Here we need only the subsort i of individuals which contains $form$ and any function type over individuals.

It should be stressed that the declaration of sorts, types and constants resembles the assertion of axioms: there is no way that Isabelle could check the "correctness" of these declarations. For example, it is possible to extend $\mathcal{FOL}$ with a type declaration $form : i$, thus enabling the quantification over formulae and stepping outside first-order logic.

These examples should have prepared the ground for the general scheme which is presented in a largely Isabelle-independent way.

2.1 Order-Sorted Types and Unification

In this subsection we summarize and restate some basic notions of order-sorted algebras (see, for example, [19]) in terms of types. We fix a set of *sorts* S and a set of *type constructors* T. An *(order-sorted) type signature* is a pair $(\leq, \Sigma)$ where $\leq$ is a partial order on S and $\Sigma \subseteq T \times S^* \times S$ is a set of declarations. In Isabelle-91 these two components are declared in the **sorts** and **types** sections respectively. In the sequel let $(\leq, \Sigma)$ be some arbitrary but fixed type signature and let $V = \bigcup_{s \in S} V_s$ be an S-sorted family of disjoint sets of variables. Instead of $(t, w, s) \in \Sigma$ we write $t : (w)s$. The relation $\leq$ extends to S^* in the componentwise way.

The set of *sorted types* is inductively defined as follows:

$$\frac{\alpha \in V_s}{\alpha : s} \qquad \frac{t : (\overline{s_n})s \quad \forall i.\ \tau_i : s_i}{t(\overline{\tau_n}) : s} \qquad \frac{\tau : s \quad s \leq s'}{\tau : s'}$$

In the sequel τ denotes arbitrary types, α and β stand for elements of V, and we write α_s to indicate that $\alpha \in V_s$. Note that "$\to$" is just one among many other type constructors and needs no special treatment.

In Isabelle-91, the principal operation on types is unification. It is required both for type inference and resolution. The definitions of substitutions, unifiers, complete sets of unifiers etc. are straightforward generalizations of the unsorted case and can be found in the general literature on order-sorted unification [18, 19, 10].

Order-sorted unification differs from unsorted unification in the cardinality of minimal complete sets of unifiers. In the unsorted case it is at most 1 (called *unitary*), in the order-sorted case it depends on the structure of $\leq$ and Σ and may be infinite. Isabelle-91 is restricted to unitary type signatures. Thus the principal type property for the language of terms is guaranteed and the nondeterminism of higher-order unification does not increase further. Although there are precise characterizations of unitary type signatures [22], the implementation relies on two sufficient but not necessary criteria, regularity and coregularity, which lead to particularly simple algorithms.

A type signature is *regular* if every type has a least sort. Regularity is decidable for finite signatures:

Theorem 2.1 (Smolka et al. [19]) *A signature $(\leq, \Sigma)$ is regular iff for every $t \in T$ and $w \in S^*$ the set $\{s \mid \exists w' \geq w.\ t : (w')s\}$ either is empty or contains a least element.*

Regularity is essential because order-sorted unification in non-regular signatures may be infinitary [19]. To arrive at unitary signatures we need two additional properties.

$$\begin{aligned} s_1 \sqcap s_2 &= \{s \mid s \leq s_1 \wedge s \leq s_2\} \\ D(t,s) &= \{w \mid \exists s'.\ t : (w)s' \wedge s' \leq s\} \end{aligned}$$

A partial ordering is called *downward complete* if any two elements with a lower bound have a greatest lower bound, i.e. if $s_1 \sqcap s_2$ contains a greatest element. A signature is *coregular* if for all $t \in T$ and all $s \in S$ the set $D(t,s)$ contains a greatest element. The finitary version of the following lemma goes back to Schmidt-Schauß and is also given by Smolka et al. [19].

Lemma 2.2 *Every regular, downward complete, and coregular signature is unitary.*

Isabelle-91 requires all type signatures to be regular, downward complete, and coregular, thus enforcing unitary unification. Comparing this lemma with the results of Waldmann [22] it turns out that only coregularity is unnecessarily strong. However it leads to a very simple algorithm described below. This algorithm is a specialization of known algorithms to downward complete and coregular signatures. Its correctness and completeness can easily be established by means of program transformation.

The algorithm is expressed by rewrite rules on pairs $\langle E, \theta \rangle$, where E is the list[2] of pairs $\tau \stackrel{?}{=} \tau'$, the problems yet to be solved, and θ is a substitution, the fragment of the solution computed so far. Solving such a pair means rewriting it to normal form. If the normal form is $\langle [], \theta \rangle$, then θ is the solution to the initial problem. Otherwise the initial problem has no solution.

$$\begin{array}{rcll} \langle (\alpha \stackrel{?}{=} \alpha) :: E, \theta \rangle & \Longrightarrow & \langle E, \theta \rangle & \\ \langle (\tau \stackrel{?}{=} \alpha) :: E, \theta \rangle & \Longrightarrow & \langle (\alpha \stackrel{?}{=} \tau) :: E, \theta \rangle & \text{if } \tau \notin V \\ \langle (\alpha_s \stackrel{?}{=} \tau) :: E, \theta \rangle & \Longrightarrow & \langle \theta'(E), W(\tau, s) \circ \theta' \circ \theta \rangle & \text{if } \alpha \text{ does not occur in } \tau \\ & & & \text{and } \theta' = \{\alpha \mapsto \tau\} \\ \langle (t(\overline{\tau_n}) \stackrel{?}{=} t(\overline{\tau'_n})) :: E, \theta \rangle & \Longrightarrow & \langle [\tau_1 \stackrel{?}{=} \tau'_1, \ldots, \tau_n \stackrel{?}{=} \tau'_n] @ E, \theta \rangle & \end{array}$$

The only interesting case is the variable-term one. In contrast to unsorted unification it is not possible simply to map α_s to τ because τ might not be of sort s. However, there may exist a *weakening*, i.e. a sort-decreasing mapping from V to V, which forces the sort of τ down to s. $W(\tau, s)$ computes the most general weakening θ such that $\theta(\tau)$ is of sort s. If there is a weakening at all, coregularity implies the existence of a most general one.

[2]Standard ML list notation is used.

Identifiers	x			Sorts	s		
Expressions	e	$=$	x	Types	τ	$=$	$\alpha_s \mid t(\tau_1, \ldots, \tau_n)$
		$\mid$	$(e_0\ e_1)$	Type schemes	σ	$=$	$\tau \mid \forall \alpha_s.\sigma$
		$\mid$	$\lambda x.e$				
		$\mid$	**let** $x = e_0$ **in** e_1				

Figure 15: Mini-ML expressions and types

The presentation of W is simplified by bringing it into tail recursive form. The call $\overline{W}(ps, \theta)$, where ps is a list of type-sort pairs and none of the variables in the domain of θ occur in ps, computes the most general instance θ' of θ such that $\theta'(\tau) : s$ for every pair (τ, s) in ps.

$$\begin{array}{rcl} W(\tau, s) & = & \overline{W}([(\tau, s)], \{\}) \\ \overline{W}([], \theta) & = & \theta \\ \overline{W}((\alpha_s, s') :: ps, \theta) & = & \left\{ \begin{array}{ll} \overline{W}(ps, \theta) & \text{if } s \leq s' \\ \overline{W}(\theta'(ps), \theta' \circ \theta) & \text{where } \theta' = \{\alpha \mapsto \beta_{glb(s,s')}\} \text{ and } \beta \text{ is new} \end{array} \right. \\ \overline{W}((t(\overline{\tau_n}), s) :: ps, \theta) & = & \overline{W}([(\tau_1, s_1), \ldots, (\tau_n, s_n)]@ps, \theta) \quad \text{where } (s_1, \ldots, s_n) = dom(t, s) \end{array}$$

$\overline{W}$ relies on the two partial functions glb and dom which are defined as follows: $glb(s_1, s_2) = s$ if $max(s_1 \sqcap s_2) = \{s\}$ and $dom(t, s) = w$ if $max(D(t, s)) = \{w\}$; otherwise they are undefined. Due to downward completeness and coregularity both functions can be stored in pre-computed tables. Table lookup at undefined positions results in failure of the unification algorithm.

2.2 Terms and Type Inference

Isabelle is based on the intuitionistic fragment of Church's Higher-Order Logic [1] and hence its terms are typed λ-terms. The grammar for "raw", i.e. as yet untyped, terms is shown in Figure 15. Let us call this language *Mini-ML* although it lacks the pattern matching of its namesake [2]. It goes beyond what is implemented in Isabelle by featuring a **let**-clause. Dealing with this richer language prepares the ground for the applications described in Section 3. To cope with **let**-expressions, the language of types introduced in the previous subsection has to be enriched with a universal quantifier, leading to the distinction between unquantified *types* and quantified *type schemes* due to Damas and Milner [4]. The rules of type inference, shown in Figure 16, are almost identical to the system DM' due to Clément et al. [2]. A is a finite mapping from identifiers to type schemes and $FV(\tau, A)$ denotes those type variables in τ which do not occur free in A. The only difference is the definition of generic instantiation. In contrast to unsorted polymorphism, where type schemes may be instantiated by arbitrary types, order-sorted polymorphism requires that the instantiation is sort-correct:

$$\forall \alpha_{s_1}, \ldots, \alpha_{s_n}.\tau \ \succeq \ \tau[\tau_1/\alpha_{s_1}, \ldots, \tau_n/\alpha_{s_n}]$$

if each τ_i is of sort s_i w.r.t. the type signature, which, as usual, remains implicit and fixed. Unsorted polymorphism can be recovered by having just a single sort, say $\top$, and exactly one declaration $t : (\top, \ldots, \top)\top$ for each type constructor.

The above inference rules can be turned into a type inference algorithm simply be interpreting them as Horn clauses and using Prolog technology. In particular this means that guessing

ASM	$$\frac{A(x) \succeq \tau}{A \vdash x : \tau}$$
APP	$$\frac{A \vdash e_0 : \tau \to \tau' \qquad A \vdash e_1 : \tau}{A \vdash (e_0\ e_1) : \tau'}$$
ABS	$$\frac{A + \{x \mapsto \tau\} \vdash e : \tau'}{A \vdash \lambda x.e : \tau \to \tau'}$$
LET	$$\frac{A \vdash e_0 : \tau \qquad FV(\tau, A) = \{\alpha_{s_1}, \ldots, \alpha_{s_k}\} \qquad A + \{x \mapsto \forall \overline{\alpha_{s_k}}.\tau\} \vdash e_1 : \tau'}{A \vdash \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1 : \tau'}$$

Figure 16: Type inference for Mini-ML

the right instantiation in ASM is replaced by unification. More precisely, generic instantiation simply peels off all universally quantified type variables and replaces them by *logical* variables in the Prolog sense. The required instantiation is found by order-sorted unification of types during resolution. Having restricted ourselves to unitary type signatures, there is at most one most-general unifier and hence principal types exist.

In Isabelle, type inference takes place w.r.t. some *theory* containing sorts, types and constants. The **sorts** and **types** section determines the type signature and the **consts** declarations the initial environment A.

2.3 Higher-Order Unification

We now turn from the static analysis to Isabelle's equivalent of execution, higher-order resolution. The details of this process, which is the heart of Isabelle's inference engine, is described elsewhere [15]. Suffice it to say that its central component is an implementation of Huet's higher-order unification procedure [8]. In contrast to type inference, which extended very smoothly to order-sorted polymorphism, the same cannot be said for higher-order unification. The reason is that type inference in Isabelle-90 already dealt with unsorted polymorphism, whereas its unification procedure was a direct implementation of Huet's algorithm which is designed for simply typed terms. The difficulties encountered in extending it to cope with type variables were solely due to polymorphism in its most basic form, i.e. the presence of type variables; the order-sorted twist was again orthogonal and remained hidden in the type unification algorithm. Hence we concentrate on unsorted polymorphism in the sequel.

The basic problem arises because terms may now contain type variables. The syntax of terms given in Figure 15 is an impoverished version of the actual Isabelle implementation where every variable and constant is tagged with its type, which may contain type variables. This type information is essential during unification when both term and type variables need to be instantiated. A simple example demonstrates the difficulty.

Let τ be some base type and α a type variable, let $c : \tau$ be a constant and $G : \alpha$ and $F : \alpha \to \tau$ be two term variables. The following matching problem is given:

$$F(G) \stackrel{?}{=} c \tag{1}$$

Let us first look at (some of) the infinitely many solutions to this problem. Every instantiation of α must be of the form

$$\tau_1 \to \cdots \to \tau_n \to \tau'$$

for suitable types $\tau_1, \ldots, \tau_n$ and τ', where τ' is not a function type. Without any assumptions about α we get the solution $F = \lambda x.c$. In terms of Huet's algorithm this is the solution obtained after a single imitation step. In fact, for any instantiation of α where $\tau' \neq \tau$, this is the only solution.

If we also assume that $\tau' = \tau$, we get the following infinite set of solutions, indexed by n:

$$F = \lambda x.x(H_1(x), \ldots, H_n(x)), \quad G = \lambda x_1, \ldots, x_n.c$$

The H_i are new free function variables.

Further instantiations of α yield yet more solutions. For example $\alpha = \tau \to \tau$ alone has an infinite set of independent solutions:

$$F = \lambda x.x^k(c), \quad G = \lambda x.x$$

is a solution for any natural number k, where x^k is the k-fold composition of x.

This shows quite drastically that type variables introduce a new degree of freedom into unification problems. Different type instantiations give rise to completely independent sets of solutions with greatly varying cardinality (finite vs. infinite). This is not in itself surprising, but it raises the question how this new freedom should be dealt with. One impractical solution is to enumerate all possible type instantiations. Isabelle-91, just like λProlog [12], is more pragmatic: only the simplest type instantiation is tried. This approach is obviously incomplete. In the above example only the two solutions $\{F \mapsto \lambda x.c\}$ and $\{F \mapsto \lambda x.x, G \mapsto c, \alpha \mapsto \tau\}$ are found. In general the problem occurs in the "projection" step of Huet's algorithm. Let the unification problem

$$\lambda x_1, \ldots, x_k.F(t_1, \ldots, t_n) \stackrel{?}{=} t$$

be given where $F : \tau_1 \to \cdots \to \tau_n \to \tau$ is a free variable, the head of t is a bound variable or constant, $\tau_i = \tau_1^i \to \cdots \to \tau_m^i \to \tau^i$ and neither τ nor τ^i are function types. Then F can be a projection on its ith argument provided the result types of τ_i and τ coincide. If neither τ_i nor τ are type variables, their result types are τ^i and τ and need only be unified. If however, for example, τ^i is a type variable, it could be instantiated by any function type with the same result type as τ, leading to the the infinite branching shown in example (1) above. Isabelle-91 unifies τ^i and τ regardless of whether they are type variables or not and produces the single projection substitution $\{F \mapsto \lambda y_1, \ldots, y_n.y_i(\ldots)\}$.

A full description of higher-order unification in Isabelle-91 can be found in [13] where it is also shown that the algorithm is complete for all unification problems whose solutions do not involve the instantiation of type variables by function types. This implies that first-order logic can be used without incompleteness because the sort system prevents type variables from ranging over function types. In higher-order logic, however, there are examples of proofs that are not found automatically because of the above incompleteness. Such situations occur infrequently and can always be remedied by user-supplied explicit type instantiations.

3 Applications to Programming Language Type Systems

Although order-sorted polymorphism was initially designed to express restricted polymorphism in Isabelle, it subsequently turned out to be equally suitable as a type system for the functional programming languages Standard ML and Haskell. The following subsections look at this application in more detail. Familiarity with the programming languages themselves is assumed. Both Standard ML and Haskell are complex languages. We analyze only a fragment of their core by reducing them to Mini-ML as introduced in Section 2. This reduction is obviously feasible for the term language, but it remains to be seen how both type systems can be reduced to order-sorted polymorphism. The benefits of this reduction are a uniform type inference algorithm and sound criteria for the existence of principal types.

3.1 Standard ML

Order-sorted polymorphism is ideally suited to model Standard ML's [11] *equality types*. They partition the set of all types into those that admit equality and those that do not. The following type signature yields a precise definition of this distinction:

$$\begin{array}{rcl} \mathcal{ML}_e & = & \textbf{sorts}\ \top \\ & & \qquad e < \top \\ & & \textbf{types}\ \texttt{bool} : e \\ & & \qquad \texttt{int} : e \\ & & \qquad \_*\_ : (\top,\top)\top \\ & & \qquad \_*\_ : (e,e)e \\ & & \qquad \_\texttt{->}\_ : (\top,\top)\top \\ & & \qquad \texttt{ref} : (\top)e \\ & & \textbf{consts}\ \_\texttt{=}\_ : \alpha_e\ \texttt{->}\ \alpha_e\ \texttt{->}\ \texttt{bool} \end{array}$$

This expresses quite succinctly that booleans, integers, and all reference types admit equality, products admit equality iff both components do, and no function type admits equality. Hence e is exactly the set of equality types according to Standard ML. Similarly, the notion of generalization $\succeq$ is precisely the one in Standard ML. Therefore type inference with equality types is just a special case of type inference in Mini-ML w.r.t. $\mathcal{ML}_e$. It is a straightforward exercise to check that $\mathcal{ML}_e$ (and the extensions below) satisfy the requirements of Lemma 2.2. Hence every typable term has a principal type. In terms of Standard ML, $\mathcal{ML}_e$ forms part of the "initial static basis" of the language. It remains to be seen how $\mathcal{ML}_e$ changes with the definition of new types.

The sorts and their ordering $\leq$ remain fixed as in **sorts** above. Hence type signatures can be identified with the set of declarations Σ. The latter is initially just the set of declarations shown in **types** above but is enriched with every new `datatype` declaration

$$\texttt{datatype}\ (\overline{\alpha_n})t\ \texttt{=}\ C_1\ [\texttt{of}\ \tau_1(\overline{\alpha_n})]\ \texttt{|}\ \ldots\ \texttt{|}\ C_k\ [\texttt{of}\ \tau_k(\overline{\alpha_n})]$$

where the portions enclosed in square brackets are optional. Assuming that we start with a signature Σ, the above `datatype` declaration automatically adds the declaration $t : (\top,\ldots,\top)\top$ to Σ. It remains to be determined whether t *admits equality* in the sense of the Standard ML [11], i.e. whether $t : (e,\ldots,e)e$ is also a valid declaration. The Definition of Standard ML is not completely formal in this respect but suggests the following interpretation:

The type constructor t admits equality provided for each i

- either C_i is a constant, i.e. `of` τ_i is omitted,
- or the type $\tau_i(\alpha_e^1,\ldots,\alpha_e^n)$ is of sort e w.r.t. $\Sigma \cup \{t : (\top,\ldots,\top)\top,\ t : (e,\ldots,e)e\}$.

As an example let us look at the definition of the (predefined) datatype `list`:

$$\texttt{datatype } \alpha \texttt{ list = nil | cons of } \alpha \texttt{ * (}\alpha\texttt{ list)}$$

In this case $\tau_2(\alpha_e) = \alpha_e$ `*` $(\alpha_e$ `list`$)$ is of sort e w.r.t. $\Sigma' = \Sigma \cup \{\texttt{list} : (\top)\top,\ \texttt{list} : (e)e\}$ provided Σ contains `_*_` $: (e,e)e$. Hence `list` admits equality and Σ' becomes the new type signature.

Standard ML's other extension to ordinary polymorphism, *imperative types*, cannot be explained with order-sorted polymorphism alone and is consequently not discussed here. However, it does not seem difficult to recast Tofte's treatment of imperative types [20] in an order-sorted framework.

3.2 Haskell

The programming language Haskell [7] generalizes Standard ML's equality types using the notion of *type classes* [21]. Below I sketch the reduction of type classes to order-sorted polymorphism which is treated in detail in [14]. The main idea is that classes correspond to sorts, `class` declarations establish the subsort relationship, and `instance` declarations determine the arities of type constructors. In the sequel let $\mathcal{C}$ be the set of all class names and let C denote a member of $\mathcal{C}$.

A closer look at Haskell reveals that type variables may belong to any finite number of classes. Hence sorts really correspond to finite, non-empty sets of classes $\mathcal{F}(\mathcal{C})$. A sort $\{C_1, \ldots, C_n\}$ can be viewed as the intersection of all the classes C_1 up to C_n, i.e. a "virtual" class that inherits the capabilities of all the constituent classes. Any subclass hierarchy, i.e. partial ordering $\preceq$ on $\mathcal{C}$, induces a unique ordering $\leq$ on sorts as follows:

$$S_1 \leq S_2 \;\Leftrightarrow\; \forall s_2 \in S_2\ \exists s_1 \in S_1.\ s_1 \preceq s_2.$$

This formalizes the intuition that a subsort must subsume all those classes contained in a supersort. Mathematically speaking, $(S, \leq)$ is the free lower semi-lattice induced by the poset $(\mathcal{C}, \preceq)$. The set S can either be viewed as the quotient $\mathcal{F}(\mathcal{C})/_{\cong}$, where $\cong\ =\ \leq \cap \geq$ ($\leq$ is only a pre-order on the whole of $\mathcal{F}(\mathcal{C})$!), or as the restriction of $\mathcal{F}(\mathcal{C})$ to those non-empty subsets of $\mathcal{C}$ whose elements are pairwise incomparable w.r.t. $\preceq$. The *glb* operation on S is simply union.

Programs in "Mini-Haskell", the fragment of Haskell described in [14], are of the form $d_1; \ldots; d_n; e$, where each d_i is a `class` or `instance` declaration and e is a Mini-ML expression. As a first step, the declarations are eliminated in two stages:

1. Extract a triple $(\preceq, \Delta, A)$, where $\preceq$ is a partial order on $\mathcal{C}$, Δ is a set of type declarations of the form $t : (S_1, \ldots, S_n)C$, and A is an environment, i.e. a mapping from identifiers to types.

2. Turn this into a proper order-sorted type signature $(\leq, \Sigma)$ and an environment A by defining $\leq$ as above and

   $$\Sigma = \{t : (W_1 \cup \ldots \cup W_k)\{C_1, \ldots, C_k\} \mid \forall i.\ (t : (W_i)C_i) \in \Delta\}$$

   where $\cup$ extends to lists of sets in the componentwise way.

Assuming we start out with $(\preceq, \Delta, A)$,

$$\texttt{class } \mathit{context} \texttt{ => } C\,\alpha \texttt{ where } x \texttt{ :: } \tau(\alpha)$$

yields the enriched triple

$$(\preceq \cup \{(C, C') \mid (C'\,\alpha) \in context\}, \Delta, A + \{x \mapsto \forall\alpha_{\{C\}}.\tau(\alpha_{\{C\}})\})$$

whereas

$$\texttt{instance}\ context\ \texttt{=>}\ C\ \ (t\,\alpha_1\ \dots\ \alpha_n)\ \ \texttt{where}\ x\ \texttt{=}\ e$$

yields

$$(\preceq, \Delta \cup \{t : (S_1, \dots, S_n)C\}, A)$$

where $S_k = \{C \mid (C\,\alpha_k) \in context\}$, $k = 1 \dots n$.

Type inference on a program $d_1; \dots; d_n; e$ proceeds as follows:

1. The d_i are processed one by one and a corresponding triple $(\preceq, \Delta, A)$ is built up.

2. The type of e is inferred in the environment A w.r.t. the type signature $(\leq, \Sigma)$.

In a real implementation the transition from $(\preceq, \Delta)$ to $(\leq, \Sigma)$ would never occur. Instead order-sorted unification would be modified to work on $(\preceq, \Delta)$ directly.

The existence of principal types in Mini-Haskell is more delicate than in Standard ML but is also implied by Lemma 2.2. The poset $(S, \leq)$ is obviously downward complete as it is a lower semi-lattice. Regularity of $(\leq, \Sigma)$ follows from the construction of Σ from Δ. Coregularity is a consequence of two context conditions in Section 4.3.2 of the Haskell report:

> "A type may not be declared as an instance of a particular class more than once in the same scope."

and a requirement on instance declarations which amounts to the following: if $C \succeq C'$ and $(t : (W')C') \in \Delta$ then there is a $W \geq W'$ such that $(t : (W)C) \in \Delta$. The derivation of coregularity from these two conditions is straightforward but tedious.

There are two complications of type inference in Haskell that the above treatment does not cover. Firstly, there are well-typed expressions which are semantically ambiguous. The prime example is `[] == []`, where `[]`: `list` α_{Eq} and `==`: α_{Eq} `->` α_{Eq} `-> bool`. Since the type of `[]` is too polymorphic to determine which instance is required, the expression is at best ambiguous or, if there are no instances of `Eq` at all, even meaningless. The type system by Wadler and Blott [21] yields the type `Eq` α `=> bool` which reveals this ambiguity: the type variable α occurs in the context but not in the type `bool` itself. In Mini-Haskell we simply infer `bool`. The reason is that we tag type variables with their sorts rather than use a context to record these associations. I conjecture that the latter device could be used in Mini-Haskell as well. In the full paper [14] the ambiguity of `[] == []` is discovered by a translation into Mini-ML.

Secondly, there is the so called "monomorphism restriction" [7, Section 4.4.2] which means that **let**-bound identifiers *must not be used at more than one distinct overloading.* For example, let $C \succ C_1, C_2$, let $t_1 : C_1$ and $t_2 : C_2$ be two distinct types, and let `f: int ->` α_C, `1000: int`, `g:` t_1 `->` t_2 `-> int`. Then the expression `let x = f 1000 in g x x` is illegal. The intention is to avoid hidden sources of inefficiency. In the above example the expression `f 1000` would need to be computed twice, once for each type t_1 and t_2. Whether to impose this restriction or not is clearly a matter of language design. The rules for Mini-Haskell do not enforce it. However it can easily be added by modifying the LET-rule. In the above example `f 1000` has type α_C and hence `x` has type $\forall\alpha_C.\alpha_C$. It suffices to ban this generalization in certain cases. In order to detect those cases we distinguish a predefined class $\top$, the class of all types. Generalization should now be restricted to type variables of sort $\{\top\}$. If there are no other classes, Mini-Haskell degenerates to Mini-ML.

4 Conclusion

We have looked at a flexible and convenient type system for the specification of object-logics with simple type systems. It offers an extension of ML-style polymorphism and automatic type inference. Where applicable, it makes life considerably easier. The concept of order-sorted polymorphism has applications beyond Isabelle involving type systems for programming languages and computer algebra. The former is sketched in Section 3, the latter can be found in an article by Comon et al. [3] where order-sorted polymorphism (although not under this name) is further enriched with subtypes. The resulting system is very expressive but type inference becomes undecidable.

Acknowledgements

Lawrence Paulson provided valuable comments on a draft version of the paper.

References

[1] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.

[2] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proc. ACM Conf. Lisp and Functional Programming*, pages 13–27, 1986.

[3] H. Comon, D. Lugiez, and P. Schnoebelen. A rewrite-based type discipline for a subset of computer algebra. *J. Symbolic Computation*, 11:349–368, 1991.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. 9th ACM Symp. Principles of Programming Languages*, pages 207–212, 1982.

[5] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. Principles of Programming Languages*, pages 52–66, 1985.

[6] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. 2nd IEEE Symp. Logic in Computer Science*, pages 194–204, 1987.

[7] P. Hudak and P. Wadler. Report on the programming language Haskell. Version 1.0, April 1990.

[8] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[9] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[10] J. Meseguer, J. Goguen, and G. Smolka. Order-sorted unification. *J. Symbolic Computation*, 8:383–413, 1989.

[11] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[12] G. Nadathur. *A Higher-Order Logic as the Basis for Logic Programming.* PhD thesis, University of Pennsylvania, Philadelphia, 1987.

[13] T. Nipkow. Higher-order unification, polymorphism, and subsorts. In *Proc. 2nd Int. Workshop Conditional and Typed Rewriting Systems.* LNCS ???, 1991.

[14] T. Nipkow and G. Snelting. Type classes and overloading resolution via order-sorted unification. In *Proc. 5th ACM Conf. Functional Programming Languages and Computer Architecture*, 1991.

[15] L. Paulson. The foundation of a generic theorem prover. *J. Automated Reasoning*, 5:363–397, 1989.

[16] L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[17] L. Paulson and T. Nipkow. Isabelle tutorial and user's manual. Technical Report 189, University of Cambridge, Computer Laboratory, 1990.

[18] M. Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* LNCS 395, 1989.

[19] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2*, pages 297–367. Academic Press, 1989.

[20] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[21] P. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc.* In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 60–76, 1989.

[22] U. Waldmann. Unification in order-sorted signatures. Technical Report 298, Fachbereich Informatik, Universität Dortmund, 1989.

Well-founded Induction and Program Synthesis Using Proof Plans

Colin Phillips[‡]
University of Edinburgh

June 1991

Abstract

The Mathematical Reasoning Group (MRG) in the Department of Artificial Intelligence at Edinburgh University has developed an approach to the verification and synthesis of programs using proof plans - high level descriptions of proofs, which control the automatic search for a proof. Constructive mathematics is employed, enabling programs to be derived using the proofs as programs technique. The proofs in question employ various forms of induction and techniques have been developed for automating the search for the form of induction appropriate to a given proof.

In this paper I will discuss a modification of this methodology which involves using well-founded induction and the automated search for the appropriate well-founded relation. This is being implemented in a version of Martin-Löf's type theory extended by Nordström's Acc Type. I shall be comparing this approach with that previously employed in the Mathematical Reasoning Group.

1 Introduction

The Mathematical Reasoning Group (MRG) in the Department of Artificial Intelligence at Edinburgh University has developed an approach to the automatic construction of proofs by induction using proof plans. A plan has two constituents, a tactic and a method. A tactic is a computer program for applying rules of inference. A method is meta-logical specification of a tactic. Two key proof plans are those for *rippling* and for *middle-out reasoning*. The rippling plan involves the rewriting of the induction conclusion to make it resemble the induction hypothesis. The existential plan uses middle-out reasoning to postpone the selection of an existential witness. Using these ideas techniques have been developed to infer the form of induction that is needed to prove a given theorem.

In this paper I will discuss a modification of this methodology which involves using well-founded induction and the automated search for the appropriate well-founded relation. This has two advantages over existing techniques.

[‡]colinp@aipna.ed.ac.uk. This research was partly supported by ESPRIT Basic Research Action "Logical Framework" grant 3245.

Firstly, at present Clam - the proof planner - simply adds schemes of induction to its repertoire as required while the modification I propose uses only a single schema, well-founded induction, thus obtaining greater generality.

Secondly, the MRG is concerned not only with the automated construction of proofs but also with the synthesis of programs using the proofs as programs technique. As a proof is constructed in Oyster an extract term is built up which contains the computational information in the proof. Unfortunately the extract term may contain computationally redundant information. If we use the Acc Type then the extract term will not contain such information and will be the program we require. 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.

However Clam is a highly successful theorem proving system, so our first concern is to insure that the attaining of these benefits still allows us to employ existing methods - in particular rippling and middle-out reasoning. In fact there already exists a mechanism in Clam which can be used to enable rippling to occur with well-founded induction - rippling-across - which I will describe. I will also outline a form of middle-out-reasoning suitable for use with well-founded induction.

2 Proof Plans and Middle-Out Reasoning

The background to this paper is the work of the MRG at Edinburgh. I shall therefore begin by giving an outline of that work. For further information see [1].

The MRG at Edinburgh University is involved in program verification and synthesis by theorem proving. It employs an implementation of Martin-Lö f's type theory, the Oyster proof development system [4] which is essentially a Prolog version of NUPRL [3]. It is used in conjunction with Clam - a proof planner.

Clam was designed as a mechanism for automatically deriving proofs in Oyster. The emphasis has been on proofs by induction where the key problem is to identify the appropriate scheme of induction and the induction variable. The basic idea of a plan is that it seeks to capture a common structure identifiable in a set of proofs. A plan is a tree of methods. A method is a tactic and a meta-logical specification of that tactic.

2.1 Associativity of Plus

Consider, as an example, the proof of the associativity of plus. Putting aside the base case we have:

$$x + (y + z) = (x + y) + z$$

$$\begin{array}{lrcl}
\vdash & s(x) + (y + z) & = & (s(x) + y) + z \\
\vdash & s(x + (y + z)) & = & s(x + y) + z \\
\vdash & s(x + (y + z)) & = & s((x + y) + z) \\
\vdash & x + (y + z) & = & (x + y) + z
\end{array}$$

At each step an instance of the successor function is moved out from an initial position to one less nested in the equation. When both instances of the successor function are in the outermost position they can be deleted leaving a copy of the induction conclusion.

Each step can be justified by an appeal to a rewrite rule. Thus we use:

$$\begin{aligned} \boxed{s(\underline{U})} + V &\Rightarrow \boxed{s(\underline{U+V})} \\ \boxed{s(\underline{U})} = \boxed{s(\underline{V})} &\Rightarrow U = V \end{aligned}$$

These rules are called wave rules. The process of successively rewriting the equation using such rules is called rippling. The final appeal to the induction hypothesis is called fertilization. The boxes designate wave fronts. The part of the argument inherited from the induction hypothesis is underlined. Rippling is a mechanism for overcoming the mismatch between induction hypothesis and conclusion to enable fertilization to take place.

So we can annotate our example as follows:

$$\begin{aligned} & x + (y + z) = (x + y) + z \\ \vdash \quad & \boxed{s(\underline{x})} + (y + z) \quad = \quad (\boxed{s(\underline{x})} + y) + z \\ \vdash \quad & \boxed{s(\underline{x + (y + z)})} \quad = \quad \boxed{s(\underline{x + y})} + z \\ \vdash \quad & \boxed{s(\underline{x + (y + z)})} \quad = \quad \boxed{s(\underline{(x + y) + z})} \\ \vdash \quad & x + (y + z) \quad = \quad (x + y) + z \end{aligned}$$

The above wave rules we have used are of the form:

$$\begin{aligned} F(\boxed{S(\underline{U})}) &\Rightarrow \boxed{S(\underline{F(U)})} \\ F(\boxed{S(\underline{U_1})}, \boxed{S(\underline{U_2})}) &\Rightarrow F(U_1, U_2) \end{aligned}$$

These are instances of a general form. Such rules rewrite terms so as to produce a closer match between the goal and the induction hypothesis.

Rippling and fertilization are implemented by rippling and fertilization methods. Clam operates on a meta-level representation of the proof sequent, and uses meta-level reasoning on the methods to select an appropriate combination of tactics.

2.2 Prime Factorisation Theorem

The meta-level representation of the proof can contain meta-variables which act as place holders for expressions whose identity has yet to be determined. Consider an example, the prime factorisation theorem. Suppose we want Clam to synthesise an appropriate induction scheme for this proof. The specification is:

$$\forall x{:}nat^+.\ \exists xl{:}list(prime).\ prod(xl) = x$$

For convenience I shall use familiar notation and not that of type theory or Oyster.

Setting aside the base case we can represent the step case with a metavariable.

$$\begin{array}{c} x' : nat^+, \\ \exists xl' : list(prime)\ prod(xl') = x' \\ \vdash \exists \boxed{xl} : list(prime) prod(\boxed{xl}) = \boxed{X} \end{array}$$

We use capitals Prolog fashion to represent meta-variables.

The meta-variable corresponds to a universally quantified variable and its instantiation will determine the form of induction needed. We annotate to indicate potential wave fronts.

We assume the following wave rules:

$$\begin{array}{rcl} (\boxed{HD :: \underline{TL}}){:}list(type) & \Rightarrow & \boxed{HD{:}type \wedge \underline{TL{:}list(type)}} \\ prod(\boxed{HD :: \underline{TL}}) & \Rightarrow & \boxed{HD \times \underline{prod(TL)}} \\ \boxed{U \times \underline{V}} = \boxed{U \times \underline{W}} & \Rightarrow & V = W \end{array}$$

We can use them in sequence to ripple out thus:

$$\begin{array}{lll} \ldots & \vdash & \boxed{P : prime \wedge \underline{XL'{:}list(prime)}} \wedge prod(\boxed{P :: \underline{XL'}}) = X \\ \ldots & \vdash & \boxed{P : prime \wedge \underline{XL'{:}list(prime)}} \wedge \boxed{P \times \underline{prod(XL')}} = X \\ \ldots & \vdash & \boxed{P : prime \wedge \underline{XL'{:}list(prime)}} \wedge prod(XL') = X' \end{array}$$

where $P :: XL'/XL$ and $P \times X'/X$.

Fertilization succeeds with instantiations: xl'/XL' and x'/X' leaving subgoal $P : prime$. So XL is $P :: xl'$ and X is $P \times x'$.

The induction scheme suggested is:

$$\frac{\vdash\ P(1) \qquad p{:}prime,\ x'{:}nat,\ P(x') \vdash P(p \times x')}{\vdash \forall x{:}nat.\ P(x)}$$

The above is an example of *middle-out reasoning.* The use of meta-variables at the start of a proof enables us to look ahead and see how the rippling will precede and hence to determine the appropriate induction scheme. For further details see [1].

3 Computational Redundancy

The remainder of this paper will deal with extensions to the above work resulting from an implementation in Oyster of well-founded induction. Such an extension seems natural given the program of synthesising induction schemes. At present induction schemes are added to Clam, as they are shown to be required for the solution of a problem by middle-out reasoning. But why not simply have a single form of induction - well-founded induction - and use middle-out reasoning to determine the relation and the set needed for a particular example.

The precise choice of an implementation was guided however by another consideration. Clam is built on top of Oyster which is an implementation of constructive mathematics. As a consequence of this it can be used not only to automate the construction of proofs but also to synthesise programs. This was in fact a motive underlying the use of constructive mathematics. In Martin-Löf's system the notion of a proof is ambiguous - we can speak of a proof as a method for making the truth of a proposition evident or we can think of a proof as a proof object or program containing the computational content of the proofs (in the first sense) of the corresponding propositions. In type theory we can think of true propositions as types inhabited by appropriate proof-objects. In Oyster - as in NUPRL - proof objects are called extract terms and are automatically constructed with the corresponding proof.

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. A version of the axiom of choice is as follows:

$$(\forall n : nat)(\exists m : nat)(R(n,m)) \rightarrow \\ (\exists f : nat \rightarrow nat)(\forall n : nat)(R(n, f(n)))$$

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 an existential claim must be backed by a witness. 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 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.

In practice it is the introduction rule for the existential quantifier that causes the trouble. $\exists x : A.F(x)$ has as its proof object a pair - the first is an object having the property F, while the second is a proof that it has that property. In the case of the axiom of choice only the first object is of interest to us. One way of dealing with this problem is to introduce into the theory sets which are not fully presented - subsets. These are in fact available in both Oyster and Clam. We write $\{x : A|F(x)\}$ for the set of objects with the property F. This only has as inhabitants objects with the attribute F. However the use of subsets in its turn raises difficulties. Consider a well known problem, the root problem. We have a function which has the value 0 for some argument and we wish to compute such an argument. So we may write as specification:

$$(\forall f : \{g : nat \rightarrow nat|\exists x : nat.g(x) = 0\}) \\ \{n : nat|f(n) = 0\}$$

Suppose we had for an input h a proof object for $\exists x : nat.g(x) = 0$. Then from this we would be able to extract both a root and a proof that it is a root. We would then be in a position to use subset introduction. But since we are using subsets this direct proof is unavailable.

4 The Acc Type

Nordström has proposed an extension to Martin-Löf's type theory which he calls the Acc type [5]. Following Nordström I use the notation $a : Acc(A, \prec)$ to mean that a is a member of the well-founded part of A with respect to the relation $\prec$. Note that there may be elements of A which are not in the well-founded part. We think of well-foundedness as involving the absence of infinite descending chains. The introduction rule is to the effect that an element is in the well-founded part of a set with respect to a relation if every element "less than" that element is in the well-founded part with respect to that relation. The elimination rule is the principle of well-founded induction.

These rules have been implemented in Oyster in sequent calculus form. In Oyster we prove goals by refinement. So the problem of proving a particular goal of the form $H_1, \ldots \vdash G$, where $H_1, \ldots$ is a list of assumptions and G is a theorem, is reduced to that of proving a list of subgoals $SG_1, \ldots$. Our goal must be derivable from these subgoals by use of one of the refinement rules.

Our introduction rule (simplified) states that in order to prove a goal of the form:

$$H \vdash X{:}acc(A, \prec)$$

We need to prove:

$$\begin{gathered} X{:}A \\ Y{:}A, Y \prec X \vdash Y{:}acc(A, R) \end{gathered}$$

The intuition here is that if all the elements that precede X are accessible then so is X.

The elimination rule (again simplified) states that in order to prove a goal of the form:

$$H \vdash SPEC(X)$$

We need to prove:

$$\begin{gathered} W : A, \\ U : \forall V{:}\{V' : A | V' \prec W\} SPEC(V) \\ \vdash SPEC(W) \\ \text{(where X:acc(A,R) in H))} \end{gathered}$$

The extract term for the goal will be $wo\_ind(X, [W, U, F])$ where F is the extract term for $SPEC(W)$.

We evaluate this as follows. Suppose that X evaluates to $X*$. Then the result of evaluating $wo\_ind(X, [W, U, F])$ is F with W and U replaced by $X*$ and $\lambda Y.(wo\_ind(Y, [W, U, F]))$ respectively.

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 the 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. It is in order to effect this 'information hiding ' that we use a subset in the elimination rule above. We illustrate with a simple example - quicksort.

(There are a number of further interesting examples of proofs involving the Acc type in Saaman and Malcolm [7]).

5 Quicksort

To illustrate the way in which one can use the Acc Type to solve the problem of computational redundancy we will consider an example - quicksort.

The specification for the quicksort program is:

$$\forall k : list(nat)\{l : list(nat)|perm(k,l) \wedge ordered(l)\}$$

We define two auxiliary functions - $filter\_left$ and $filter\_right$ which partition the tail of a list $h :: t$ into two sublists, a list of elements of t less than h and a list of elements of t greater than or equal to t.

If we take the specification as a theorem to be proved in Oyster we must determine the appropriate relation - in this case we will take "$l \prec m$" to mean that that the length of l is less than the length of m. Next we invoke a tactic which given that relation generates two subgoals. One is:

$$\begin{array}{c} m : list(nat) \\ \vdash m{:}acc(list(nat), \prec) \end{array}$$

In effect the proof of this establishes that lists are well-founded under the $\prec$ relation, where we take the measure for this to be the length of a list. Such proofs employ the standard induction scheme for natural numbers. The second subgoal is:

$$\begin{array}{c} m : list(nat) \\ g : \forall n : \{n' : list(nat)|n' \prec m\}\{l : list(nat)|perm(n,l) \wedge ordered(l)\} \\ \vdash \{l' : list(nat)|perm(m,l') \wedge ordered(l')\} \end{array}$$

We do a case case split, according to whether or not m is nil.

We consider the case where m is nil. Clearly nil is an ordered permutation of nil so this case is trivial.

We take m to be of the form $(h :: t)$. We abbreviate:

$$\begin{array}{c} filter\_left(h,m)\ to\ LEFT\ m \\ filter\_right(h,m)\ to\ RIGHT\ m \end{array}$$

We do an elimination on n with $LEFT\ m$. We also do an elimination on n with $RIGHT\ m$. These steps require us to show that both introduced lists are less than m. We then use the induction hypothesis to obtain ordered permutations of both filters, say $g(LEFT\ m)$ and $g(RIGHT\ m)$. We then do an introduction with $g(LEFT\ m) <> h :: g(RIGHT\ m)$. We prove that this is an ordered permutation of m.

Consider now the extract term.

The termination proof establishes that $k : acc(list(nat), \prec)$. We abbreviate our goal to $SPEC(k)$.

The extract term for the goal will be:

wo_ind(k,[m,g,f])

Where f is the extract term for the conclusion of the second subgoal, that is:

$$g(LEFT\ m) <> h :: g(RIGHT\ m)$$

The result of evaluating $wo\_ind(k, [m, g, f])$ is f with m replaced with k and g with:

$$\lambda y.(wo\_ind(y, [m, g, f]))$$

That is:

$$wo\_ind(LEFT\ k, [m, g, f]) <> (hd\ k) :: (wo\_ind(RIGHT\ k, [m.g, f]))$$

I have used familiar notation in place of Oyster terminology. The extract term obtained from the proof of the above in Oyster is precisely the program we want; and if we evaluate it as applied to a list of natural numbers, it will output a sorted list of those numbers.

6 The Root Problem revisited

We recall the specification:

$$(\forall f : \{g : nat \to nat | \exists x : nat.g(x) = 0\})\{n : nat | f(n) = 0\}$$

To solve this we need to use a set I defined as:

$$I = \{y : nat | \forall t : nat.t < y \to \neg f(t) = 0\}$$

Our induction step is:

$$\begin{array}{c} p : I \\ g : \forall q : \{q' : I | q' = succ(p)\}\{n : nat | f(n) = 0\} \\ \vdash \{n : nat | f(n) = 0\} \end{array}$$

We do a case split with $f(p) = 0$. If this is true then p is our witness. Otherwise we have $\neg(f(p) = 0)$. But p is in I and, since p is not a root, $succ(p)$ is also in I and we have our goal. We need to find an item which is in I. 0 will do. We must prove that $0 : Acc(I, pred)$. This is trivial as I is finite.

So have as the extract term for the induction step:

if f(p) = 0 then p else g(succ p)

The extract term for the goal will be:

wo_ind(0,[p,g,if f(p) = 0 then p else g(succ p)])

The evaluation of this is:

If f(0) = 0 then 0 else wo_ind(succ(0),[p,g,h]).

Again I have used familiar notation instead of Oyster's.

7 Rippling-across

If we extend Oyster with the Acc Type we have provided ourselves with the means for modifying Clam so that we employ a single form of induction - well-founded induction. Using the Acc Type will also guarantee that the extract term from a proof will contain no computationally redundant information. However Clam is a highly successful theorem proving system, so our first concern is to insure that the attaining of these benefits still allows us to employ existing methods - in particular rippling and middle-out reasoning.

In this connection the first point to observe is that the principle of well-founded induction gives a schema for induction which does not contain wave-fronts so how is the rippling to begin? In fact however there is an experimental mechanism in Clam which is able to deal with this problem - that of rippling-across. The basic idea here is due to A.Steven's, see [2].

This was introduced to deal with proofs involving destructor style induction. One's first thought here is that one will obtain wave fronts in the induction hypothesis so one can ripple forwards in the induction hypothesis rather than backwards in the induction conclusion. However for reasons which need not concern us here a somewhat more complicated method is needed. In rippling-across we create in the induction conclusion wave-fronts and anti-wave-fronts. The latter neutralise the wave fronts in the induction hypothesis.

Consider an example - the associativity of append. The specification is:

$$\forall l, m, n : list(nat). l <> (m <> n) = (l <> m) <> n$$

I will assume that the induction will be on the variable l. We replace l with a metavariable in the induction hypothesis to indicate that it is a potential wave front. So we have:

$$\begin{gathered} l, m, n : listnat, \\ L <> (m <> n) = (L <> m) <> n, \\ \vdash l <> (m <> n) = (l <> m) <> n \end{gathered}$$

Where we need to show that any substitution for L is less than l by some appropriate measure.

We now use a destructor style definition of *append*.

$$\neg(X = nil) \rightarrow X <> Y \quad = \quad \boxed{hd(X) :: \underline{\boxed{(tl(\underline{X}))} <> Y}}$$

This definition suggests rewriting the induction step in the following manner:

$$\begin{array}{rcl} \boxed{tl(\underline{l})} <> (m <> n) & = & (\boxed{tl(\underline{l})} <> m) <> n \\ \vdash \boxed{hd(l) :: \underline{\boxed{(tl(\underline{l})} <> (m <> n)}} & = & (\boxed{hd(l) :: \underline{\boxed{(tl(\underline{l}))} <> m}}) <> n \end{array}$$

This leads to the choice of the length of the list as the appropriate well-founded relation, since the length of the tail of non-empty list is less than that of the entire list. The wave fronts now introduced in the induction conclusion "cancel" those in the assumption.

$$\begin{array}{rcl} tl(l) <> (m <> n) & = & (tl(l) <> m) <> n \\ \vdash \boxed{hd(l) :: \underline{(tl(l) <> (m <> n))}} & = & (\boxed{hd(l) :: \underline{(tl(l) <> m)}}) <> n \end{array}$$

We ripple out one step using:

$$(\boxed{l :: \underline{m}}) <> n \rightarrow \boxed{l :: \underline{(m <> n)}}$$

To obtain:

$$\begin{array}{rcl} tl(l) <> (m <> n) & = & (tl(l) <> m) <> n \\ \vdash \boxed{hd(l) :: \underline{(tl(l) <> (m <> n))}} & = & \boxed{hd(l) :: \underline{((tl(l) <> m) <> n)}} \end{array}$$

We can now fertilize to complete the proof.
So we see that it is possible to use the rippling method with well-founded induction.

8 Prime Factorisation

In general we are interested in using middle-out reasoning to find a program satisfying a specification of the form:

$$\forall x : A\{y : B|R(x, y)\}$$

So we are looking for a program $PROG$ such that $PROG(x) : \{y : B|R(x,y)\}$. In order to use the induction hypothesis in the case of a particular x of type A it is necessary that x should be "reducible" in some, yet to be determined, sense. If x is such a reducible object then there will be an object of the form $g(x_1, \ldots x_n, y_1, \ldots y_n)$, call it $x*$ such that $x = x*$. (In fact there are cases where the relation between x and $x*$ is not identity but I shall not consider this complication.) The objects x_i must be of type A and less than x by some well founded measure. We can then assume the existence of $PROG(x_i)$ for any such x_i and use this construction in obtaining $PROG(x)$.

In general we will need, in connection with a particular example, a lemma to the effect that $x*$ exists for any reducible x, and that the constituents of $x*$ have appropriate properties. In the case of an object x_i this will be to the effect that $x_i \prec x$, so we can then use the objects x_i to eliminate on the induction hypothesis. The key problem is to identify the relation in which the x_i stand to x.

To see how existing methods in Clam can be adapted to deduce this function let us look again at the prime factorisation theorem. The induction hypothesis is:

$$\begin{array}{c} p : nat^+ \\ \forall x : \{x' : nat^+|x' < p\}\{xl : list(prime)|product(xl) = x'\} \\ \vdash \{pl : list(prime)|product(pl) = p\} \end{array}$$

We assume as before (but with base cases):

$$nil{:}list(type) \tag{1}$$

$$(\boxed{HD :: \underline{TL}}){:}list(type) \Rightarrow \boxed{HD{:}type \wedge \underline{TL{:}list(type)}} \tag{2}$$

$$prod(nil) = s(0) \tag{3}$$

$$prod(\boxed{HD :: \underline{TL}}) \Rightarrow \boxed{HD \times \underline{prod(TL)}} \tag{4}$$

$$\boxed{U \times \underline{V}} = \boxed{U \times \underline{W}} \Rightarrow V = W \tag{5}$$

We can rewrite our goal as:

$$\vdash PL : A \wedge product(PL) = P$$

The meta-variable PL indicates that we are looking for an appropriate witness. The meta-variable P indicates that we may substitute for p expressions containing wave-fronts. We now look at possible instantiations for P. If rippling is to take place we need a match with the left hand side of a wave rule. (1),(2) and (4) are of the wrong form. (3) gives us a solution for $p = s(0)$. (5) is a possible match. So we try rewriting P as $U \times W$. But now we need a rule to rewrite $prod(PL)$ as $U \times V$. (2) will do this provided PL is of the form $HD :: TL$.Then we can ripple out (just as before):

$$\begin{array}{ll}
\ldots & \vdash \boxed{HD : prime \wedge \underline{TL{:}list(prime)}} \wedge prod(\boxed{HD :: \underline{TL}}) = U \times W \\
\ldots & \vdash \boxed{HD : prime \wedge \underline{TL{:}list(prime)}} \wedge \boxed{HD \times \underline{prod(TL)}} = U \times W \\
\ldots & \vdash \boxed{HD : prime \wedge \underline{TL{:}list(prime)}} \wedge prod(TL) = W
\end{array}$$

As a result of the second step we can impose on U the condition that it be prime. Consider the third step. As a general strategy we should leave invoking the induction hypothesis for as long as possible since this will only work with reducible objects and we have yet to determine what these are. We try (3) again for a match. This gives us one with $TL = nil$ and $W = s(0)$. This gives a solution if p is prime. We now need to invoke the induction hypothesis. We instantiate TL with $PROG(V)$. This requires us to prove that $W \prec P$. Our solution is then $U :: PROG(V)$ with U prime. So we deduce that a reducible number in this example to be a composite number - a number which is neither $s(0)$ nor prime.

What is the relation we are looking for? The above reasoning suggests a necessary condition:

$$x \prec y \rightarrow \exists p{:}prime.(y = p \times x)$$

But we also need a relation with respect to which the set we are interested in is well-founded. This means that the relation must not be reflexive. So we have:

$$x \prec y \leftrightarrow \exists p{:}prime.(y = p \times x) \wedge \neg(x = y)$$

We now have the lemma we require. It is to the effect that any composite number y is the product of a prime number p and a number x such that $x \prec y$, where $\prec$ is defined as above. We have to prove this lemma and show that the positive natural numbers are well-founded with respect to this relation.

We conclude that it is possible to perform middle-out reasoning with well-founded induction using methods which are similar to those already in existence in Clam.

9 Conclusion

We have looked at a way of extending Oyster with the Acc Type and Clam with appropriate methods for well-founded induction. The extension is a natural one, enabling rippling and middle-out-reasoning to be used in a wider context. In this sense it confirms the soundness of those basic ideas. The implementation of the Acc Type in Oyster provides empirical verification that it can be used to eliminate computational redundant information, a result which is of some significance as the elimination of such information is a problem which many researchers engaged in synthesising programs confront. For example see [6]

One of the advantages of using the Acc Type in Clam is that it gives one a slightly different perspective on how to deal with the various kinds of eureka steps involved in a proof. A perspective that one might consider more natural for a programmer trying to synthesis a program from a specification, rather than a mathematician trying to automate the proof of a theorem. So one looks for a proof of termination rather than the proof of the correctness of a scheme of induction. This connects up the work of the MRG with that of researchers automating proofs of well-foundedness. Possibly this will be a source of new ideas enabling further extensions to the basic ideas already employed.

References

[1] A.Bundy, A.Smaill,J.Hesketh. "Turning Eureka Steps into Calculations in Automatic Program Synthesis" In proceedings of UK IT, 1990

[2] Bundy, A. et al. "Rippling: A Heuristic for Guiding Inductive Proofs." Forthcoming

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

[4] 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.

[5] B. Nordström. "Terminating General Recursion." In preparation (1987).

[6] C. Paulin & B. Werner. "Extracting and Executing Programs in the Inductive Constructions System: a Progress Report." Proceedings of the First Annual Workshop on Logical Frameworks.

[7] 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

Towards an Algebraic Framework for Defining Sequential Logics*

David Pym
University of Edinburgh
Scotland, U.K.
dpym@lfcs.ed.ac.uk

July 1991

(Intuitionistic/minimal) type-theoretic logical frameworks are inadequate in several respects:

- They are intuitionistic — weak logics are difficult to represent;
- The notion of logic as proposition-object together with proof-object is present, but too restrictive and not analysed (it is so restricted that it *is* natural deduction);
- There is no analysis of sequent calculus;
- There is no analysis of proof reduction;
- There is no *general* access to a model theory;
- There is no analysis of the *notion* of a framework — characterization of encoding *etc.*.

We propose a framework based on early ideas of Kelly, some recent developments of Hagino, the hierarchical sensibilities of Generalized Type Systems and Avron's analysis of consequence relations. To this end we propose notions of algebra and dialgebra for indexed functors. Then fibred adjunctions permit us to define generalizations of Hagino-like datatypes and (syntactic) specification language. Starting from an axiomatization of some primitive monoidal connectives, one can obtain, uniformly, Gentzen-type axiomatizations of such inductively defined logics.

Such a framework provides the possibility a quite general analysis of (dependent) type theories, including logically weak theories (*cf.* intuitionistic linear logic-as-linear dependent type theory: Bellin, Miller, Plotkin, Pym, 1991), considered as sequent calculus together with a notion of proof-object, from the point of view of finitary algebraic presentation.

These proposals provide an analysis of the notion of a framework and access to a general notion of Model Theory.

One of our (long term) aims is to provide theorems such as: general conditions for the presence of cut-elimination (*cf.* early work of Kelly), natural deduction systems, connection (matrix) calculi, *etc.*.

Related work: P. Schroeder–Heister; P. Aczel; E. Moggi; J. Power; C. Wells; E. Ritter and others.

*This work is supported by U.K. Science and Engineering Research Council grant GR/G 58588, "Logical and Semantical Frameworks"; and by European Community ESPRIT Basic Research Action 3245, "Logical Frameworks: Design, Implementation and Experiment"

An Interpretation of Kleene's Slash in Type Theory

Jan M. Smith
Department of Computer Science, University of Göteborg/Chalmers
S-412 96 Göteborg, Sweden

May 1991

Abstract

Kleene introduced the notion of slash to investigate the disjunction and existence properties under implication for intuitionistic arithmetic. In this paper Kleene's slash is translated to type theory. Besides translations of Kleene's results, the main application of the slash in type theory is that conditions are given for a typable term, containing free variables, to have a normal form beginning with a constructor.

1 Introduction

The disjunction and existence properties, that is, $\vdash A \vee B$ implies $\vdash A$ or $\vdash B$ and $\vdash \exists x A(x)$ implies $\vdash A(t)$ for some term t, respectively, were first proved for intuitionistic arithmetic by Kleene [6] using a modification of recursive realizability. Harrop [5] extended Kleene's result by also considering derivations depending on assumptions. Harrop proved

$$C \vdash A \vee B \text{ implies } C \vdash A \text{ or } C \vdash B \tag{1}$$

$$C \vdash \exists x A(x) \text{ implies } C \vdash A(t) \text{ for some term } t \tag{2}$$

when C is a closed formula not containing any strictly positive occurrences of $\vee$ and $\exists$; such a formula is called a Harrop formula.

In [7] Kleene gives much simplified proofs of the disjunction and existence properties as well as (1) and (2) for a more extensive class of formulas than the Harrop formulas. I will in this paper translate Kleene's method to type theory and thereby obtain conditions on a typable term to have a canonical value also when it may contain free variables.

Tait [13] introduced a powerful method for proving normalization of typed terms which was adopted by Martin-Löf [8] to prove normalization of derivations in natural deduction. These ideas are behind Prawitz' [11, 12] suggestion of a general notion of validity for derivations. In Hallnäs [4] there is an extension of Prawitz notion of validity which is closely related to slash in type theory. In fact, Tait's computability method, when applied to type theory, can be seen as a special case of the translation below of slash.

I will use Martin-Löf's type theory [9, 10] and, in order not to obscure the ideas, I will not treat universes; to include universes involve the same additional complications as when proving normalization, using Tait's method, for Martin-Löf's type theory with universes.

Friedman [3] has defined slash for higher order logic and I expect no problems in translating that notion of slash to impredicative type theories like Coquand and Huet's Calculus of Constructions [2]. However, since in the Calculus of Constructions types are defined in the usual higher order way, it is not clear to me how some of the results below should be interpreted in this context.

2 Kleene's slash

The modification of recursive realizability in Kleene [6], used for the proofs of the disjunction and existence properties, consists in adding derivability conditions to the definition of realizability. The crucial observation in Kleene [7] is that the proofs of the disjunction and existence properties in [6] do not rely on any recursion theoretic elements, which, hence, can be left out, thereby giving very simple proofs of these properties.

Let Γ be a list of closed formulas and C a closed formula. The relation $\Gamma \mid C$, "Γ slashes C", is defined by induction on the number of logical constants in C by the clauses

1. If A is an atomic formula, $\Gamma \mid A$ if $\Gamma \vdash A$.
2. $\Gamma \mid A \vee B$, if $\Gamma \mid A$ or $\Gamma \mid B$.
3. $\Gamma \mid A \& B$, if $\Gamma \mid A$ and $\Gamma \mid B$.
4. $\Gamma \mid A \supset B$, if $\Gamma \vdash A \supset B$, and $\Gamma \mid A$ implies $\Gamma \mid B$.
5. $\Gamma \mid \forall x A(x)$, if $\Gamma \vdash \forall x A(x)$ and $\Gamma \mid A(\overline{n})$ for each numeral $\overline{n}$.
6. $\Gamma \mid \exists x A(x)$, if $\Gamma \mid A(\overline{n})$ for some numeral $\overline{n}$.

This definition of slash is a simplification, due to Aczel [1], of Kleene's original definition. The main result in [6] for the slash is the following theorem.

Theorem 1 *Let Γ be a list of closed formulas and A a closed formula. Assume that $\Gamma \mid C$ for each C in Γ. Then $\Gamma \vdash A$ implies $\Gamma \mid A$.*

The proof is by induction on the length of the derivation $\Gamma \vdash A$. From this theorem it easy to obtain (1) and (2) for closed formulas C such that $C \mid C$, using the observation that $\Gamma \mid A$ implies $\Gamma \vdash A$; since $C \mid C$ holds if C is a Harrop formula, this implies Harrop's result for (1) and (2).

3 Translation of Kleene's slash to type theory

In arithmetic, slash is a relation between a list of closed formulas and a closed formula; this will in type theory be translated to a relation between a context Γ and a judgement $t \in A$ in the context Γ.

The definition of $\Gamma \mid t \in A$ is made by induction on the derivation $\Gamma \vdash A$ set by the clauses

1. $\Gamma \mid t \in A + B$, if $\Gamma \vdash t \in A + B$, $\Gamma \vdash t = \mathsf{inl}(a) \in A + B$ for some term a such that $\Gamma \mid a \in A$, or $\Gamma \vdash t = \mathsf{inr}(b) \in A + B$ for some term b such that $\Gamma \mid b \in B$.
2. $\Gamma \mid t \in \Pi(A, B)$, if $\Gamma \vdash t \in \Pi(A, B)$, there exists a $b(x)$ for which $\Gamma, x \in A \vdash b(x) \in B(x)$, such that, for all terms a, $\Gamma \mid a \in A$ implies $\Gamma \mid b(a) \in B(a)$, and $\Gamma \vdash t = \lambda x.b(x) \in \Pi(A, B)$.

3. $\Gamma \mid t \in \Sigma(A, B)$, if $\Gamma \vdash t \in \Sigma(A, B)$, there exist terms a and b such that $\Gamma \mid a \in A$ and $\Gamma \mid b \in B(a)$, so that $\Gamma \vdash t = \langle a, b\rangle \in \Sigma(A, B)$.

4. $\Gamma \mid t \in \mathsf{Id}(A, a, b)$, if $\Gamma \vdash t \in \mathsf{Id}(A, a, b)$, $\Gamma \mid a \in A$, $\Gamma \mid b \in A$, $\Gamma \vdash a = b \in A$, and $\Gamma \vdash t = \mathsf{id}(a) \in \mathsf{Id}(A, a, b)$.

5. $\Gamma \mid t \in \mathsf{N}$, if $\Gamma \vdash t \in \mathsf{N}$ and $\Gamma \vdash t = \overline{n} \in \mathsf{N}$ for some numeral $\overline{n}$.

6. $\Gamma \mid t \in \mathsf{N}_k$, if $\Gamma \vdash t \in \mathsf{N}_k$ and $\Gamma \vdash t = n_k \in \mathsf{N}_k$ for some n, $0 \leq n < k$, $k = 0, 1, \ldots$.

It follows directly from the definition that $\Gamma \mid t \in A$ implies $\Gamma \vdash t \in A$. Note that, since $\perp$ is defined to be N_0, the second condition in the definition of $\Gamma \mid t \in \mathsf{N}_0$ gives that there is no term t such that $\Gamma \mid t \in \perp$ holds. This is different from arithmetic where $\perp$ is defined to be $0 = 1$ and, hence, $\Gamma \mid \perp$ for any inconsistent Γ. If we consider the absurdity $\perp_{\mathrm{M}}$ of minimal logic, we can extend the definition of slash by

7. $\Gamma \mid t \in \perp_{\mathrm{M}}$, if $\Gamma \vdash t \in \perp_{\mathrm{M}}$.

The rules for $\perp_{\mathrm{M}}$ differs from the rules for $\perp$ in that there is no elimination rule for $\perp_{\mathrm{M}}$.

From the definition of slash, we get the following theorem.

Theorem 2 *If $\Gamma \mid t \in A$ then there exists a canonical value can of the set A such that $\Gamma \vdash t = can \in A$.*

A canonical value is a term on normal form beginning with a constructor. The formulation of Martin-Löf's type theory we are using is the one in which $a = b \in A$ means definitional equality; so $\Gamma \vdash t = can \in A$ implies that t can be computed to the value *can*.

Theorem 3 *Let Δ be a context $x_1 \in D_1, \ldots, x_m \in D_m(x_1, \ldots, x_{m-1})$ and assume that for each i, $0 < i \leq m$, there exists a term d_i such that $\Gamma \mid d_i \in D_i(d_1, \ldots, d_{i-1})$. Then $\Delta \vdash a(x_1, \ldots, x_m) \in A(x_1, \ldots, x_m)$ implies $\Gamma \mid a(d_1, \ldots, d_m) \in A(d_1, \ldots, d_m)$.*

The proof is by induction on the length of the derivation $\Delta \vdash a(x_1, \ldots, x_m) \in A(x_1, \ldots, x_m)$. As a special case of this theorem we get the translated version of theorem 1.

Theorem 4 *Let Γ be a context $x_1 \in C_1, \ldots, x_n \in C_n(x_1, \ldots, x_{n-1})$ and assume that for each i, $0 < i \leq n$, there exists a term c_i such that $\Gamma \mid c_i \in C_i(c_1, \ldots, c_{i-1})$. Then $\Gamma \vdash a(x_1, \ldots, x_n) \in A(x_1, \ldots, x_n)$ implies $\Gamma \mid a(c_1, \ldots, c_n) \in A(c_1, \ldots, c_n)$.*

Theorems 2 and 3 give that if $t \in A$ can be derived in the the empty context, then t can be computed to a canonical value of A. This result can also be obtained by a normalization proof, using Tait's computability method; actually, if Γ is empty then the definition of $\Gamma \mid t \in A$ is the same as the definition of Tait's computability predicate $Comp_A(t)$.

Because of the strong rules for Σ in Martin-Löf's type theory, the existence property is trivially satisfied in any context Γ since, by Σ-elimination, $\Gamma \vdash t \in \Sigma(A, B)$ implies $\Gamma \vdash fst(t) \in A$ and $\Gamma \vdash snd(t) \in B(fst(t))$. Note, however, that the term $fst(t)$ cannot in general be computed to a canonical value. The disjunction property cannot be proved under such general conditions. Corresponding to Kleene's result for disjunction in arithmetic we have the following corollary to theorem 4.

Corollary 1 *Let C be a set for which there exists a term $c(x)$ such that $x \in C \mid c(x) \in C$. Then $x \in C \vdash t(x) \in A + B$ implies $x \in C \vdash a(x) \in A$ for some term $a(x)$ or $x \in C \vdash b(x) \in B$ for some term $b(x)$.*

In arithmetic, disjunction can be expressed by using the existential quantifier; in Martin-Löf's type theory $+$ can in a similar way be expressed by Σ. But it is easy to see that the existence property obtained by the strong Σ-elimination does not imply the disjunction property.

4 Harrop sets

The interest of Theorem 4 depends on for what sets we have $x \in C \mid c(x) \in C$ and on the complexity of the term $c(x)$. In arithmetic the main examples of formulas C such that $C \mid C$ are the Harrop formulas. A Harrop formula is defined to be a formula not containing any strictly positive occurrences of $\vee$ and $\exists$. Alternatively, the Harrop formulas can be inductively defined by (i) atomic formulas are Harrop formulas, (ii) if A and B are Harrop formulas then $A \& B$ is a Harrop formula, (iii) if B is a Harrop formula and A an arbitrary formula then $A \supset B$ is a Harrop formula, and (iv) if A is a Harrop formula then $\forall x A$ is a Harrop formula.

In type theory, we define the notion corresponding to Harrop formulas in an arbitrary context. The Harrop sets H_Γ in the context Γ are inductively defined by

(i) N_1, the one element set, is in H_Γ,
(ii) if $\Gamma \mid a \in A$, $\Gamma \mid b \in A$, and $\Gamma \vdash a = b \in A$ then $\mathsf{Id}(A, a, b)$ is in H_Γ,
(iii) if A is in H_Γ and $B(x)$ in $H_{\Gamma, x \in A}$ then $\Sigma(A, B)$ is in H_Γ,
(iv) if A is a set in Γ and $B(x)$ in $H_{\Gamma, x \in A}$ then $\Pi(A, B)$ is in H_Γ.

Note that the sets in (i)-(iv) are those with exactly one constructor. By clause (iii), $A \times B$ is a Harrop set if A and B are Harrop sets and, by clause (iv), $A \to B$ is Harrop set if B is a Harrop set. Because of clause (iii) it is possible for a Harrop set to contain a strictly positive occurrence of a set of the form $\Sigma(A, B)$, but that requires A to be Harrop set; so there cannot be any strictly positive occurrences of sets of the form $\Sigma(\mathsf{N}, B)$ corresponding to an existentially quantified proposition on the natural numbers. If we consider minimal logic we can extend the Harrop sets by

(v) $\perp_{\mathrm{M}}$ is in H_Γ.

We cannot, however, include $\perp$ in the definition of Harrop sets since we would then not have the following theorem, corresponding to that, in arithmetic, $C \mid C$ when C is a Harrop formula.

Theorem 5 *If C is a Harrop set in the context Γ then there exists a term $c(x)$ such that*

$$\Gamma,\ x \in C \mid c(x) \in C$$

The term $c(x)$ can be recursively constructed by

(i) $\Gamma \mid 0_1 \in \mathsf{N}_1$,
(ii) *if* $\Gamma \mid a \in A$, $\Gamma \mid b \in A$, *and* $\Gamma \vdash a = b \in A$ *then* $\Gamma \mid \mathsf{id}(a) \in \mathsf{Id}(A, a, b)$,
(iii) *if* $\Gamma,\ x \in A \mid a(x) \in A$ *and* $\Gamma,\ x \in A,\ y \in B(x) \mid b(x, y) \in B(x)$ *then*
$$\Gamma,\ z \in \Sigma(A, B) \mid \langle a(\mathit{fst}(z)), b(\mathit{fst}(z), \mathit{snd}(z))\rangle \in \Sigma(A, B),$$
(iv) *if* $\Gamma,\ x \in A,\ y \in B(x) \mid b(x, y) \in B(x)$ *then*
$$\Gamma,\ z \in \Pi(A, B) \mid \lambda x.b(x, \mathsf{apply}(z, x)) \in \Pi(A, B),$$
(v) $\Gamma,\ x \in \perp_{\mathrm{M}} \mid x \in \perp_{\mathrm{M}}$.

A Harrop set has at most one constructor, which intuitively explains why it is possible to construct a term $c(x)$ beginning with a constructor of the set C by only using the variable $x \in C$.

In the proof of theorem 5 we need the following lemma.

Lemma 1 *Let Γ be a context $x_1 \in C_1, \ldots,\ x_n \in C_n(x_1, \ldots, x_{n-1})$, let $C(x_1, \ldots, x_n)$ be a Harrop set in Γ, and let the term $c(x_1, \ldots, x_{n-1}, x)$ be constructed according to* (i) - (v) *in theorem 5. Let Δ be a context for which there exist terms $a_1, \ldots,\ a_n$ and a such that $\Delta \vdash a_i \in C_i(a_1, \ldots, a_{i-1})$, $1 \leq i \leq n$, and $\Delta \vdash a \in C(a_1, \ldots, a_n)$. Then $\Gamma,\ x \in C(x_1, \ldots, x_n) \mid c(x_1, \ldots, x_n, x) \in C(x_1, \ldots, x_n)$ implies $\Delta \mid c(a_1, \ldots, a_n, a) \in C(a_1, \ldots, a_n)$.*

Proof. The proof of the lemma is by induction on the definition of C in H_Γ. We will look at the case when C is of the form $\Pi(A, B)$. By the recursive definition of c we know that c is $\lambda x.b(x, \mathsf{apply}(z, x))$ for some term b such that $\Gamma,\ x \in A,\ y \in B(x) \mid b(x, y) \in B(x)$ (1). Assume that $\Delta \vdash d \in \Pi(A, B)$ and $\Delta \mid a \in A$. We then obtain $\Delta \vdash a \in A$ (2) and, hence, $\Delta \vdash \mathsf{apply}(d, a) \in B(a)$ (3). From (1), (2), (3) and the induction hypothesis, we get $\Delta \mid b(a, \mathsf{apply}(d, a)) \in B(a)$. Hence, by the definition of slash, $\Delta \mid \lambda x.b(x, \mathsf{apply}(d, x)) \in \Pi(A, B)$.

Proof of theorem 5. The proof of theorem is by induction on the definition of C in H_Γ. We will look at the case when C is of the form $\Pi(A, B)$. By induction hypothesis we have $\Gamma,\ x \in A,\ y \in B(x) \mid b(x, y) \in B(x)$ (1). Assume $\Gamma,\ z \in \Pi(A, B) \mid a \in A$, which gives $\Gamma,\ z \in \Pi(A, B) \vdash a \in A$ (2) and $\Gamma,\ z \in \Pi(A, B) \vdash \mathsf{apply}(z, a) \in B(a)$ (3). The lemma applied on (1), (2) and (3) gives $\Gamma,\ z \in \Pi(A, B) \mid b(a, \mathsf{apply}(z, a)) \in B(a)$. Hence, by the definition of slash, $\Gamma,\ z \in \Pi(A, B) \mid \lambda x.b(x, \mathsf{apply}(z, x)) \in \Pi(A, B)$.

As a corollary to theorem 3 and theorem 5 we get

Corollary 2 *Let C be a Harrop set and let $c(x)$ be constructed according to* (i)-(v) *in theorem 5. If $x \in C \vdash a(x) \in A$ then there exists a canonical value $can(x)$ of the set A such that $x \in C \vdash a(c(x)) = can(x) \in A$.*

If we use the negation of minimal logic, that is, $\neg_{\mathrm{M}} A$ is defined to be $A \to \bot_{\mathrm{M}}$, then, by theorem 5, $x \in \neg_{\mathrm{M}} A \mid \lambda y.\mathsf{apply}(x, y) \in \neg_{\mathrm{M}} A$; hence we have the following corollary.

Corollary 3 *Let C be an arbitrary set. If $x \in \neg_{\mathrm{M}} C \vdash a(x) \in A$ then there exists a canonical value $can(x)$ of the set A such that $x \in \neg_{\mathrm{M}} C \vdash a(\lambda y.\mathsf{apply}(x, y)) = can(x) \in A$.*

So, a term containing a variable $x \in \neg_{\mathrm{M}} C$ can be computed to a canonical value if the variable x is simply η-expanded.

Acknowledgements. I would like to thank Thierry Coquand and Lars Hallnäs for many long and stimulating discussions on the topic of this paper. Thanks also to Per Martin-Löf, in particular for pointing out to me that Tait's method can be seen as a special case of the above translation of slash to type theory.

References

[1] Peter Aczel. Saturated intuitionistic theories. In H. A. Schmidt, K. Schütte, and H.-J. Thiele, editors, *Contributions to Mathematical Logic*, pages 1–11. North-Holland, 1968.

[2] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[3] Harvey Friedman. Some applications of Kleene's Methods for Intuitionistic Systems. In A. R. D. Mathias and H. Rogers, editors, *Cambridge Summer School in Mathematical Logic*. Springer-Verlag, 1973.

[4] Lars Hallnäs. Partial Inductive Definitions. To appear in *Theoretical Computer Science*.

[5] R. Harrop. Concerning formulas of the types $\mathrm{A} \rightarrow \mathrm{B} \vee \mathrm{C}$, $\mathrm{A} \rightarrow (\exists x)\mathrm{B}(x)$ in intuitionistic formal systems. *Journal of Symbolic Logic*, 25:27–32, 1960.

[6] S. C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10:109–124, 1945.

[7] S. C. Kleene. Disjunction and existence under implication in elementary intuitionistic formalisms. *Journal of Symbolic Logic*, 27:11–18, 1962.

[8] Per Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In J. E. Fenstad, editor, *In Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland Publishing Company, 1971.

[9] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[10] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[11] Dag Prawitz. Towards a Foundation of a General Proof Theory. In P. Suppes et al., editor, *Logic, Methodology and Philosophy of Science IV*, pages 225–250. North-Holland, 1973.

[12] Dag Prawitz. On the Idea of a General Proof Theory. *Synthese*, 27:63–77, 1974.

[13] W. W. Tait. Intensional interpretation of functionals of finite type I. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

A Machine Checked Proof that Ackermann's Function is not Primitive Recursive*

Nora Szasz
Department of Computer Sciences
Chalmers University of Technology
and
University of Göteborg

June 1991

1 Introduction

In logic, many different languages have been suggested for the formalization of mathematics. Some of these languages are of interest to computer science, since they might be used as programming logics. As a consequence of this, editors and automatic tools for using such formalized languages have been developed. This work is an experiment with proposals arising from both these fields: Martin-Löf's set theory [ML84] as a foundational framework for mathematics, and ALF (Another Logical Framework) [ACN90] as editor and checker of a basic formalized language in which the above can be expressed.

The example we chose is the proof that Ackermann's function is not primitive recursive. The proof is mathematically simple in the sense that it only uses induction and some basic properties of the addition and the order of natural numbers. The result involves obviously also the definition of the primitive recursive functions. This definition is strongly based on tuples, a fact usually not considered in standard presentations.

In section 2 we extend the set of primitive recursive functions to internalize the notion of tuples of functions, and we prove an extension of Ackermann's result for these functions. In section 3 we present the formalization of this proof in Martin-Löf's set theory and we explain how it was written and structured using ALF. In section 4 we make some comments about the development of the proof and design decisions.

2 Primitive Recursive Functions, Tuples of Primitive Recursive Functions and Ackermann's Result

In the first part of this section, we will show how the set of primitive recursive functions is defined. Then we will define Ackermann's function and show how the proof that it grows faster than any primitive recursive function is carried out.

*This work is shorter version of the thesis with the same name, presented for the Licentiate Degree in Computer Science at Chalmers University of Technology and University of Göteborg, Sweden, 1991.

In the second part of the section, the definition of primitive recursive functions will be extended to include functions yielding values in N^n for $n>0$. This will be done by adding an appending operator for functions to the set of operators defining the primitive recursive functions. For this set of functions (called *t-primitive recursive functions*) we will state and prove an extension of Ackermann's result.

This proof will be done in a detailed way, although it is just a simple generalization of the original one, which is available in many textbooks on recursive functions theory (see [Her65]). The reason for presenting the proof here is that its structure will be used later in the formalization.

2.1 Primitive Recursive Functions and Ackermann's Result

The primitive recursive functions are built by first considering the most fundamental functions over natural numbers: the *constant* 0 function and the *successor* function. Then, the *projections* to select elements in tuples. Over these operations, a principle of definition of functions by *recursion* on one of the arguments is added, and finally the class is closed by *composition*.

Definition 1 *A function $f:N^n \rightarrow N$ is primitive recursive if and only if it is*

1. *Z, the constant function 0 of zero arguments, defined by $Z() = 0$,*
2. *the successor function S, defined by $S(x) = x+1$,*
3. *one of the projections π_n^i, defined by $\pi_n^i(x_1,...,x_n) = x_i$, for $0 < i \leq n$,*
4. *defined by composition of primitive recursive functions $h:N^m \rightarrow N$ and $g_1,...,g_m:N^n \rightarrow N$, that is, $f(x_1,...,x_n) = h(g_1(x_1,...,x_n),...,g_m(x_1,...,x_n))$,*
5. *defined by primitive recursion from primitive recursive functions $g:N^{n-1} \rightarrow N$ $(n>0)$ and $h:N^{n+1} \rightarrow N$, by* $\begin{array}{rl} f(0,x_2,...,x_n) &= g(x_2,...,x_n) \\ f(k+1,x_2,...,x_n) &= h(f(k,x_2,...,x_n),k,x_2,...,x_n). \end{array}$

This set of functions describes a large variety of the functions used in mathematics. They are computable and total, but not all computable total functions belong to this set. Ackermann defined [Ack28] a function which is computable and total, but not primitive recursive.
Let us now define[1] a family $Ack_n:N \rightarrow N$ $(n \in N)$ of primitive recursive functions by:

$$\begin{cases} Ack_0 &= S \\ Ack_{n+1}(0) &= Ack_n(1) \\ Ack_{n+1}(y+1) &= Ack_n(Ack_{n+1}(y)) \end{cases}$$

It is easy to show by induction that for all $n \epsilon N$, Ack_n is primitive recursive. Now, define $A:N^2 \rightarrow N$ by $A(x,y) = Ack_x(y)$. This is a function that is computable and total (because every Ack_n is) but which is not primitive recursive.

This is proved by first showing that for any primitive recursive function $f:N^n \rightarrow N$ there exists a constant k_f such that $f(x_1,...,x_n) < A(k_f, \sum_{i=1}^n x_i)$ for all $x_1,...,x_n \in N$, which is done by structural induction on the definition of primitive recursive functions. Then, assuming A primitive recursive yields a contradiction: if A is primitive recursive so is the function $F:N \rightarrow N$, defined by $F(x) = A(x,x)$, and then we obtain that for all $x \in N$, $F(x) < A(k_F, x)$ holds, in particular for $x = k_F$ we get $A(k_F, k_F) = F(k_F) < A(k_F, k_F)$.

[1]The function to be defined is not Ackermann's original function, but a simplification of it that preserves its major properties [Pét35].

2.2 Tuples of Primitive Recursive Functions and the Extension of Ackermann's Result

In order to completely formalize the notion of primitive recursive function, we need to consider some details that are usually left unattended. When considering the definition of a function f by composition of h and $g_1, ..., g_m$, we will note that there is an implicit operation of *tupling* made over $g_1, ..., g_m$, i.e. f is defined as the composition of two functions: h and $(g_1,...,g_m)$, where $(g_1,...,g_m)$ is the function defined by $(g_1,...,g_m)(t) = (g_1(t),...,g_m(t))$, for $t \in N^n$. So, we must first internalize the notion of tuples and tuples of functions, and we will do this by extending the primitive recursive functions to be functions in $N^n \to N^m$, for $n, m \in N$. Instead of adding a tupling operator for functions (as described above), we will add an *appending* operator for functions, which will be based on the appending operation for tuples $\langle \, , \, \rangle$. The appending operator for functions has the same effect for our purposes as a tupling one, but it is simpler than a tupling operator in the sense that it takes a constant number of functions (two) to yield a new one (see section 4.1). This motivates the following definition:

Definition 2 *A function $f: N^n \to N^m$ $(m>0)$ is t-primitive recursive if and only if it is*

1. *Z, the constant function 0 of zero arguments, defined by $Z() = 0$,*
2. *the successor function S, defined by $S(x) = x+1$,*
3. *one of the projections π_n^i, defined by $\pi_n^i(x_1,...,x_n) = x_i$ for $0 < i \leq n$,*
4. *defined by composition of t-primitive recursive functions $h: N^k \to N^m$ and $g: N^n \to N^k$, that is, $f(x_1,...,x_n) = h(g(x_1,...,x_n))$,*
5. *defined by primitive recursion from t-primitive recursive functions $g: N^{n-1} \to N^m$ $(n>0)$ and $h: N^{n+m} \to N^m$, by:*
   $f(0, x_2,..., x_n) = g(x_2,..., x_n)$
   $f(k+1, x_2,..., x_n) = h(\langle f(k, x_2,..., x_n), (k, x_2,..., x_n)\rangle)$
6. *defined by appending t-primitive recursive functions $g: N^n \to N^{m_1}$ and $h: N^n \to N^{m_2}$ such that $m_1 + m_2 = m$, that is, $f(x_1,...,x_n) = \langle g(x_1,...,x_n), h(x_1,...,x_n)\rangle$.*

The function A can be defined in the same way as before, and an extension of Ackermann's result for t-primitive recursive functions can be formulated, by using as measure for the result of the application of a function on a tuple the addition of all its components.
So, we define the norm of a tuple $(x_1,..., x_n)$ to be:

$$\|(x_1,..., x_n)\| = \sum_{i=1}^{n} x_i.$$

Corresponding to Ackermann's result for t-primitive recursive functions is the following:
Theorem. *For any t-primitive recursive function $f : N^n \to N^m$, there exists a constant k_f such that, for all $t \in N^n$:*

$$\|f(t)\| < A(k_f, \|t\|)$$

Then, as before, assuming that A is t-primitive recursive will yield a contradiction.
In order to prove this theorem we need to prove the following properties of A:

For all $x, y \in N$:

A 1 $A(0, y) = y+1$.
Proof. By definition of A.

A 2 $A(x+1, 0) = A(x, 1)$.
Proof. By definition of A.

A 3 $A(x+1, y+1) = A(x, A(x+1, y))$.
Proof. By definition of A.

A 4 $y < A(x, y)$.
Proof. By induction on x:
For $x = 0$, we have that for all $y \in N$, $y < y+1 = A(0, y)$.
Suppose that $y < A(x, y)$ holds for all $y \in N$ (IH_1). Then, for all y, $y < A(x+1, y)$, by induction on y:
For the base case, we have $0 < 1 < A(x, 1)$ (by IH_1), but $A(x, 1) = A(x+1, 0)$, and hence, $0 < A(x+1, 0)$.
Suppose now that $y < A(x+1, y)$ (IH_2). Then, $A(x+1, y) < A(x, A(x+1, y))$ (by IH_1 for $A(x+1, y)$), and so, by IH_2, we have $y+1 \leq A(x+1, y) < A(x, A(x+1, y)) = A(x+1, y+1)$.

A 5 *For all $x, y_1, y_2 \in N$, if $y_1 < y_2$, then $A(x, y_1) < A(x, y_2)$.*
(Monotonicity in the second argument.)
Proof. It is enough to show that $A(x, y) < A(x, y+1)$ $(*)$. Then, the monotonicity follows by induction on the difference between y_1 and y_2.
We prove $(*)$ by induction on x: For $x = 0$ we have that $A(0, y) = y+1 < A(0, y+1)$, by **A4**. Else, also by **A4**, $A(x+1, y) < A(x, A(x+1, y)) = A(x+1, y+1)$.

A 6 $A(x, y+1) \leq A(x+1, y)$.
Proof. By induction on y:
For $y = 0$, we have that $A(x, 1) = A(x+1, 0)$.
Suppose now $A(x, y+1) \leq A(x+1, y)$ for all $x \in N$ (IH). Then, by **A4**, $y+1 < A(x, y+1)$ and so, $y+2 \leq A(x, y+1) \leq A(x+1, y)$ (by IH). Finally, by **A5**, $A(x, y+2) \leq A(x, A(x+1, y)) = A(x+1, y+1)$.

A 7 *For all $x_1, x_2, y \in N$, if $x_1 < x_2$, then $A(x_1, y) < A(x_2, y)$.*
(Monotonicity in the first argument.)
Proof. It is enough to show that $A(x, y) < A(x+1, y)$ and then make induction on the difference between x_1 and x_2.
This follows from $A(x, y) < A(x, y+1)$, by **A5**, and $A(x, y+1) \leq A(x+1, y)$, by **A6**.

A 8 $A(1, y) = y+2$.

Proof. By induction on y.

A 9 $A(2,y) = 2y{+}3$.
Proof. By induction on y, using **A8**.

A 10 *For all $x_1, x_2 \in N$, there exists $k^\star \in N$ such that $A(x_1, A(x_2, y)) < A(k^\star, y)$ for all $y \in N$.*
Proof.

$$\begin{aligned} A(x_1, A(x_2,y)) &< A(x_1{+}x_2, A(x_1{+}x_2{+}1, y)) && \text{(by monot. of } A\text{)} \\ &= A(x_1{+}x_2{+}1, y{+}1) && \text{(by } \mathbf{A3}\text{)} \\ &\leq A(x_1{+}x_2{+}2, y) && \text{(by } \mathbf{A6}\text{)} \end{aligned}$$

So, we can choose $k^\star_{x_1,x_2}$ to be $x_1{+}x_2{+}2$.

A 11 *For all $x_1, x_2 \in N$, there exists $k^+ \in N$ such that $A(x_1,y){+}A(x_2,y) < A(k^+, y)$ for all $y \in N$.*
Proof.

$$\begin{aligned} A(x_1,y){+}A(x_2,y) &\leq 2A(x_1{+}x_2, y) && \text{(by monot. of } A\text{)} \\ &< 2A(x_1{+}x_2, y){+}3 \\ &= A(2, A(x_1{+}x_2, y)) && \text{(by } \mathbf{A9}\text{)} \\ &< A(k^\star_{2,x_1+x_2}, y) && \text{(by } \mathbf{A10}\text{)} \end{aligned}$$

So, taking $k^+_{x_1,x_2} = k^\star_{2,x_1+x_2}$, the proposition holds.

A 12 *If $k \in N$ is such that $x < A(k,y)$, then there exists k' such that $x{+}y < A(k',y)$.*
Proof.
$y < A(0,y)$ by **A4**, and so $x{+}y < A(k,y){+}A(0,y) < A(k^+_{k,0}, y)$ (by **A11**).
Hence, we can take $k' = k^+_{k,0}$.
Now, we give the complete proof of the theorem, which is by induction on the definition of t-primitive recursive functions.
For the basic functions:

- $\|Z()\| = 0 < 1 = A(0,0) = A(0, \|(\,)\|)$.
- $\|S(x)\| = x{+}1 < x{+}2 = A(1,x) = A(1, \|(x)\|)$.
- $\|\pi^i_n(x_1,..., x_n)\| = x_i \leq \|(x_1,..., x_n)\| < A(0, \|(x_1,..., x_n)\|)$, by **A4**.

For the induction steps, we assume the corresponding induction hypothesis in each case:

- Let f be defined by composition of h and g, and let $t \in N^n$. Then,

  $$\begin{aligned} \|h(g(t))\| &< A(k_h, \|g(t)\|) && \text{(by ind. hyp. for } h\text{)} \\ &< A(k_h, A(k_g, \|t\|)) && \text{(by ind. hyp. for } g \text{ and monot. of } A\text{)} \\ &< A(k^\star_{k_h,k_g}, \|t\|) && \text{(by } \mathbf{A10}\text{)} \end{aligned}$$

- For functions defined by primitive recursion, a stronger result will be proved:

  If f is defined by primitive recursion from g and h, then there exists a constant k'_f such that, for all $t \in N^n$:

  $$\|f(t)\|{+}\|t\| < A(k'_f, \|t\|) \tag{1}$$

  By the induction hypothesis, we know that there exist k_g and k_h such that the theorem holds for g and h respectively, and so, by **A12**, there exist k'_g and k'_h such that (1) holds for g and h also. We will prove that (1) holds by taking $k'_f = k'_g{+}k'_h{+}1$.
  Let $t = (x, t_1) \in N^n$ ($n{>}0$). The proof is by induction on x.

  For $x = 0$ we have:

$$
\begin{aligned}
\|f(0,t_1)\|+\|(0,t_1)\| &= \|g(t_1)\|+\|t_1\| && \text{(by def. of } f \text{ and } \|\ \|)\\
&< A(k'_g, \|t_1\|) && \text{(by ind. hyp. for } g)\\
&< A(k'_f, \|t_1\|) && \text{(by monot. of } A \text{ and def. of } k'_f)\\
&= A(k'_f, \|(0,t_1)\|) && \text{(by def. of } \|\ \|)
\end{aligned}
$$

Assume now that $\|f(x,t')\|+\|(x,t')\| < A(k'_f, \|(x,t')\|)$ holds for all tuples of the form (x,t') (IH). Then we have that,

$$
\begin{aligned}
\|f(x{+}1,t_1)\|+\|(x{+}1,t_1)\| &= \|h(f(x,t_1),x,t_1)\|+\|(x,t_1)\|+1 && \text{(by def. of } f \text{ and } \|\ \|)\\
&\leq \|h(f(x,t_1),x,t_1)\|+\|(f(x,t_1),x,t_1)\|+1 && \text{(by def. of } \|\ \|)\\
&< A(k'_h, \|(f(x,t_1),x,t_1)\|)+1 && \text{(by ind. hyp. for } h)
\end{aligned}
$$

and so,

$$
\begin{aligned}
\|f(x{+}1,t_1)\|+\|(x{+}1,t_1)\| &\leq A(k'_h, \|(f(x,t_1),x,t_1)\|)\\
&= A(k'_h, \|f(x,t_1)\|+\|(x,t_1)\|) && \text{(by def. of } \|\ \|)\\
&< A(k'_h, A(k'_f, \|(x,t_1)\|)) && \text{(by IH and monot. of } A)\\
&\leq A(k'_f{-}1, A(k'_f, \|(x,t_1)\|)) && \text{(by monot. of } A \text{ and def. of } k'_f)\\
&= A(k'_f, \|(x,t_1)\|+1) && \text{(by } \mathbf{A3})\\
&= A(k'_f, \|(x{+}1,t_1)\|) && \text{(by def. of } \|\ \|)
\end{aligned}
$$

- Finally, if f is obtained by appending g and h, then,

$$
\begin{aligned}
\|\langle g(t),h(t)\rangle\| &= \|g(t)\|+\|h(t)\| && \text{(by def. of } \|\ \|)\\
&< A(k_g, \|t\|)+A(k_h, \|t\|) && \text{(by ind. hyp. for } g \text{ and } h)\\
&< A(k^{+}_{k_h,k_g}, \|t\|) && \text{(by } \mathbf{A11})
\end{aligned}
$$

The proof of the theorem is now complete.
To conclude, let us see that the original result for primitive recursive functions can be obtained from this one. The set of primitive recursive functions corresponds to the subset of t-primitive recursive functions that have arity $N^n \to N^1$. The basic functions are the same for both sets, and the instance of the primitive recursive schema for t-primitive recursive functions when $m = 1$ is exactly the same schema as for primitive recursive functions. In order to define a function $f : N^n \to N$ by composition of $h : N^m \to N$ and $g_1, ..., g_m : N^n \to N$, a new function $G : N^n \to N^m$ must be defined by *appending* the functions $g_1, ..., g_m$ (which is done in $m{-}1$ steps), and then, f is defined by composition of h and G.

Now, by the definition of $\|\ \|$, the result for t-primitive recursive functions, when instantiated to functions belonging to the subset of functions with arity $N^n \to N^1$, is the same result as the one for primitive recursive functions.

3 A Complete Formalization of the Proof in Martin-Löf's Set Theory

In this section, we will present a complete formalization of the proof presented above for t-primitive recursive functions. The formalization will be done in Martin-Löf's set theory. The proof was written in an implementation of the (monomorphic) set theory in ALF [ACN90] which was also used to edit and check the proof.

A complete presentation of Martin-Löf's set theory can be found in [NPS90]. We will start by recalling some notational conventions. Then we will show how we formalized the proof in

Martin-Löf's set theory. We will first present the new sets and propositions defined. Then, we will show the operations defined over these sets and the properties proved for them, together with the representation of the main lemmas of the proof. Finally, we will show how the proof was written and structured using ALF.

3.1 Notational Conventions

We refer to [NPS90] concerning Martin-Löf's set theory. We will use the same names as appear there for sets, constants, and rules.

A proposition is identified with the set of its proofs. So we will write A *prop* when we want to see the set A as a proposition, and will write A *true* whenever we know that there is an element $a \in A$.

Assumptions are of the form $x \in A$, where x is a variable and A is a set, and they are written $[x \in A]$.

The syntax we will use for expressions is the following: if e is an expression and x a variable, then $x.e$ is the abstraction of x from e. The application of an expression f to another expression a is written $f(a)$. Successive abstractions and applications will be denoted $x.y.e$ and $f(a,b)$ respectively, instead of the corresponding $x.(y.e)$ and $f(a)(b)$.

In the case of the non canonical constants, we will change the order of the arguments, putting the argument corresponding to the eliminated element as the last one (instead of putting it in the first place).

We abbreviate the proposition $Id(A,a,b)$ by $a =_A b$.

Rules have the form:

$$\frac{P_1 \quad P_2 \quad \cdots \quad P_n}{C}$$

where the the premises $P_1, P_2, \ldots, P_n$ and the conclusion C are (possibly hypothetical) judgements.

3.2 Basic Sets and Relations

In the formalization of the proof, we will need some new inductively defined sets and propositions. We will first introduce the order for natural numbers, which will be defined as an inductive binary relation. Then, we will define two more sets, the set of tuples of natural numbers, and the set of t-primitive recursive functions.

Order

To define the order for natural numbers we will first define an inductive family of sets *Less* by:

Less-formation

$$\frac{n \in N \quad m \in N}{Less(n,m)\ Set}$$

Less-introduction$_1$

$$\frac{m \in N}{z\text{-}Less(m) \in Less(0, succ(m))}$$

Less-introduction$_2$

$$\frac{n \in N \quad m \in N \quad p \in Less(n,m)}{s\text{-}Less(n,m,p) \in Less(succ(n), succ(m))}$$

The elimination rule provides an induction principle for the set defined. Once given the formation and introduction rules, elimination and equality rules can be deduced automatically, following [CP90, Dyb90]:

Less-elimination

$$\frac{\begin{array}{l} C(x,y,z)\ Set\ [x,y \in N,\ z \in Less(x,y)] \\ z(y) \in C(0, succ(y), z\text{-}Less(y))\ [y \in N] \\ s(x,y,z,u) \in C(succ(x), succ(y), s\text{-}Less(x,y,z))\ [x,y \in N,\ z \in Less(x,y), u \in C(x,y,z)] \\ n \in N \\ m \in N \\ p \in Less(n,m) \end{array}}{LessE(z,s,n,m,p) \in C(n,m,p)}$$

We will not state the equality rules, which are straightforward from the elimination and introduction rules.
With this definition, we can define the $<$ proposition, by:

$< \ \equiv\ Less.$

And from this we obtain the following rules for $<$by forgetting the proof elements:

<-formation

$$\frac{n \in N \quad m \in N}{n < m\ prop}$$

<-introduction$_1$

$$\frac{m \in N}{0 < succ(m)\ true}$$

<-introduction$_2$

$$\frac{n \in N \quad m \in N \quad n < m\ true}{succ(n) < succ(m)\ true}$$

<-elim

$$\frac{\begin{array}{l} C(x,y)\ prop\ [x,y \in N,\ x < y\ true] \\ C(0, succ(y))\ true\ \ [y \in N] \\ C(succ(x), succ(y))\ true\ \ [x,y \in N,\ x < y\ true,\ C(x,y)\ true] \\ n \in N \\ m \in N \\ n < m\ true \end{array}}{C(n,m)\ true}$$

Finally, we define $\leq$ by:

$n \leq m \equiv (n < m \vee n =_N m)$

which is a proposition under the assumptions $n, m \in N$.

Tuples of Natural Numbers
In order to represent the tuples of natural numbers, we will define a family of sets T, depending on N, in such a way that an element in $T(n)$ will be a tuple of n elements. The family is defined using the constructors *nil* and *cons*:

T-formation

$$\frac{n \,\epsilon\, N}{T(n)\ Set}$$

T-introduction$_1$

$$nil \,\epsilon\, T(0)$$

T-introduction$_2$

$$\frac{n \,\epsilon\, N \quad a \,\epsilon\, N \quad t \,\epsilon\, T(n)}{cons(a,t) \,\epsilon\, T(succ(n))}$$

T-elimination

$$\frac{\begin{array}{l} C(x,y)\ Set\ [x \,\epsilon\, N,\ y \,\epsilon\, T(x)] \\ d \,\epsilon\, C(0, nil) \\ e(y,z,u) \,\epsilon\, C(succ(x), cons(y,z))\ [x, y \,\epsilon\, N,\ z \,\epsilon\, T(x),\ u \,\epsilon\, C(x,z)] \\ n \,\epsilon\, N \\ t \,\epsilon\, T(n) \end{array}}{Telim(d,e,t) \,\epsilon\, C(n,t)}$$

t-Primitive Recursive Functions

The next set to define is the set of t-primitive recursive functions. By looking at the definition of these functions in the previous section, it is clear that this is an inductively defined set, defined by Z, S, the *projections*, *composition*, *primitive recursion* and *appending*.

There are two things to take into account when defining this set: the *arities* of the functions (domain and codomain), and their *behavior* when applied to tuples of the correct arity. The arities can be made explicit by defining a *family* of sets TPR, such that an element in $TPR(n,m)$ will represent a function in $N^n \!\to\! N^m$. The behavior of the elements of this family as functions will be defined later, when we define the meaning of each constant as a function. So, with this definition, we will only define the syntax of t-primitive recursive functions. From this definition, we will obtain an induction principle, which we will use both to define the meaning of and to prove propositions over the elements in the sets TPR.

TPR-formation

$$\frac{n \,\epsilon\, N \quad m \,\epsilon\, N}{TPR(n,m)\ Set}$$

We define a canonical constant for each possible way of constructing a t-primitive recursive function:

TPR-introduction$_1$

$$Z \,\epsilon\, TPR(0,1)$$

TPR-introduction$_2$

$$S \,\epsilon\, TPR(1,1)$$

TPR-introduction$_3$

$$\frac{n \in N \quad i \in N}{\pi(n,i) \in \mathit{TPR}(\mathit{succ}(i+n),1)}$$

Remember that π^j_m is defined only when $0 < j \leq m$. In this case, a very simple way to make this condition over the indexes hold is by forcing it by pattern matching. So, we define the new constant π in such a way that $\pi(n,i)$ is equivalent to the old $\pi^{i+1}_{i+n+1}(t)$, and thus, $0 < i+1 \leq i+n+1$ will hold for all $n, i \in N$.

TPR-introduction$_4$

$$\frac{n \in N \quad k \in N \quad m \in N \quad h \in \mathit{TPR}(k,m) \quad g \in \mathit{TPR}(n,k)}{\mathit{comp}(k,h,g) \in \mathit{TPR}(n,m)}$$

TPR-introduction$_5$

$$\frac{n \in N \quad m \in N \quad g \in \mathit{TPR}(n,m) \quad h \in \mathit{TPR}(\mathit{succ}(m+n),m)}{\mathit{pr}(g,h) \in \mathit{TPR}(\mathit{succ}(n),m)}$$

TPR-introduction$_6$

$$\frac{n \in N \quad m_1 \in N \quad m_2 \in N \quad g \in \mathit{TPR}(n,m_1) \quad h \in \mathit{TPR}(n,m_2)}{\mathit{app}(m_1,m_2,g,h) \in \mathit{TPR}(n,m_1+m_2)}$$

In these two last cases, the conditions in the definition of t-primitive recursive functions ($n > 0$ and $m_1+m_2 = m$ respectively) are again forced by pattern matching.

TPR-elimination

$$\frac{\begin{array}{l}
C(x,y,z)\ \mathit{Set}\ [x,y \in N,\ z \in \mathit{TPR}(x,y)] \\
z \in C(0,1,Z) \\
s \in C(1,1,S) \\
r(n,i) \in C(\mathit{succ}(i+n),1,\pi(n,i))\ [n,i \in N] \\
c(n,k,m,h,g,u,v) \in C(n,m,\mathit{comp}(k,h,g))\ [n,k,m \in N,\ h \in \mathit{TPR}(k,m),\ g \in \mathit{TPR}(n,k), \\
\qquad\qquad u \in C(k,m,h),\ v \in C(n,k,g)] \\
p(n,m,g,h,v,u) \in C(\mathit{succ}(n),m,\mathit{pr}(g,h))\ [n,m \in N,\ g \in \mathit{TPR}(n,m), \\
\qquad\qquad h \in \mathit{TPR}(\mathit{succ}(m+n),m),\ v \in C(n,m,g),\ u \in C(\mathit{succ}(m+n),m,h)] \\
a(n,m_1,m_2,g,h,v,u) \in C(n,m_1+m_2,\mathit{app}(m_1,m_2,g,h))\ [n,m_1,m_2 \in N,\ g \in \mathit{TPR}(n,m_1), \\
\qquad\qquad h \in \mathit{TPR}(n,m_2),\ v \in C(n,m_1,g),\ u \in C(n,m_2,h)] \\
n \in N \\
m \in N \\
f \in \mathit{TPR}(n,m)
\end{array}}{\mathit{TPRe}(z,s,r,c,p,a,n,m,f) \in C(n,m,f)}$$

In order to improve readability we will omit the variable k in $\mathit{comp}(k,h,g)$, and the variables m_1 and m_2 in $\mathit{app}(m_1,m_2,g,h)$ whenever they can be inferred from the context.

3.3 Operations and Properties

In this section, we will first define the operations on tuples that are used in the proof and the functions represented by the elements of the sets $TPR(n,m)$. Then we will show which mathematical properties involving these operations and the order and equality over natural numbers we have to prove.

Operations on Tuples

The following operations on tuples will be used in the formalization of the proof:

$hd(t) \equiv Telim(0,\ y.z.u.y,\ t)\ \epsilon\, N\ \ [n\,\epsilon\, N,\ t\,\epsilon\, T(n)]$
gives the first element of a tuple (0 in the *nil* case),

$tl(t) \equiv Telim(nil,\ y.z.u.z,\ t)\ \epsilon\, T(pred(n))\ \ [n\,\epsilon\, N,\ t\,\epsilon\, T(n)]$
gives the tail of a tuple (*nil* if the tuple is *nil*),

$[k] \equiv cons(k, nil)\ \epsilon\, T(1)\ \ [k\,\epsilon\, N]$
the constructor of a tuple with one element,

$chgfst(a,t) \equiv cons(a, tl(t))\ \epsilon\, T(succ(n))\ \ [a, n\,\epsilon\, N,\ t\,\epsilon\, T(succ(n))]$
changes the first element of the (non-empty) tuple t to a,

$\langle t_1, t_2\rangle \equiv Telim(t_2,\ y.z.u.cons(y,u),\ t_1)\ \epsilon\, T(n_1{+}n_2)\ \ [n_1, n_2\,\epsilon\, N,\ t_1\,\epsilon\, T(n_1),\ t_2\,\epsilon\, T(n_2)]$
the appending operation,

$proj^i_n(t) \equiv apply(natrec(\lambda(t_1.hd(t_1)),\ u.v.\lambda(t_1.apply(v,\ tl(t_1))), i),\ t)\ \epsilon\, N\ \ [n, i\,\epsilon\, N,$
$t\,\epsilon\, T(succ(i{+}n))]$
gives the i+1-th element of the tuple,

$\|t\| \equiv Telim(0,\ y.z.u.y{+}u,\ t)\ \epsilon\, N\ \ [n\,\epsilon\, N,\ t\,\epsilon\, T(n)]$
the norm of a tuple.

There are some remarks about these definitions:

First, note that, although *hd* and *tl* are usually defined only for non-empty tuples, we defined them as total functions. This is because in this framework all the functions are total, so, if the domain is a family of sets, the operation must be defined over all the sets of the family.

About the definition of *proj*, the same thing we did for the definition of π was done here in order to make the condition over the indexes hold. Note also that in the definition of *proj* it was necessary to use higher order functions in the following sense: with *natrec* we define an element in the set $\Pi x \epsilon T(succ(i{+}n)).\, N$, and then apply it to the tuple t to get the desired result. This is done in order to be able to make the definition of $proj^i_n(t)$ by recursion on i, on which t depends. In fact, alternative definitions of $\langle\ ,\ \rangle$ and $\|\ \|$ can be given by recursion on natural numbers using the same idea. Let us define:

$$\langle t_1, t_2\rangle_{n_1} \equiv apply(natrec(\lambda(t.t_2),\ u.v.\lambda(t.cons(hd(t),\ apply(v, tl(t))))),\ n_1),\ t_1)$$
$$\epsilon\, T(n_1{+}n_2)\ \ [n_1, n_2\,\epsilon\, N,\ t_1\,\epsilon\, T(n_1),\ t_2\,\epsilon\, T(n_2)],$$

$$\|t\|_n \equiv apply(natrec(0,\ u.v.\lambda(t_1.hd(t_1){+}apply(v, tl(t_1)))), n), t)\,\epsilon\, N\ \ [n\,\epsilon\, N, t\,\epsilon\, T(n)].$$

We prefer these definitions to the former ones (although they are more complicated) because with a definition by recursion on natural numbers some equalities are easier to prove (see section 4.2). We will neither write the variable n_1 in $\langle t_1, t_2\rangle_{n_1}$ nor n in $\|t\|_n$ (although the constants depend on them) whenever they can be inferred from the context.

Meaning of t-Primitive Recursive Functions
To complete the definition of t-primitive recursive functions, we will now define the meaning of the elements in TPR as functions. This will be done by defining, for each $n, m \in N$ and $f \in TPR(n,m)$ a constant $TPR_{fun}(n,m,f)$ in $T(n) \rightarrow T(m)$ that will behave as the t-primitive recursive function it represents, according to equations *1-6* of Definition 2 in section 2.2. The definition of TPR_{fun} is by induction on the definition of the elements in TPR, using TPR elimination. So, in order to define it, we must provide a function for each way of constructing an element in $TPR(n,m)$, under the corresponding induction hypothesis.

- For Z the constant 0 function z_f is defined:
  $z_f \equiv \lambda(t.[0]) \in T(0) \rightarrow T(1)$.
- For S we define the function s_f, which applied to a tuple with one element returns a tuple formed with the successor of that element:
  $s_f \equiv \lambda(t.[succ(hd(t))]) \in T(1) \rightarrow T(1)$.
- For $\pi(n,i)$, we use *proj* already defined for tuples, to define the function r_f:
  $r_f(n,i) \equiv \lambda(t.[proj_n^i(t)]) \in T(succ(i+n)) \rightarrow T(1) \;\; [n, i \in N]$.
- For the composition of $h \in TPR(k,m)$ and $g \in TPR(n,k)$, we must define a function which composes the function defined by h and the function defined by g. So, we define the function c_f:
  $$c_f(n,k,m,h,g,f_h,f_g) \equiv \lambda(t.apply(f_h, apply(f_g,t))) \in T(n) \rightarrow T(m) \;\; [n,k,m \in N, h \in TPR(k,m),\; g \in TPR(n,k),\; f_h \in T(k) \rightarrow T(m),\; f_g \in T(n) \rightarrow T(k)].$$
- For a function f defined by primitive recursion on g and h, we need to make induction on the first element of the tuple we apply the function. In the 0 case, the result is the function defined by g applied to the tail of the tuple. In the successor case, we need to build a new tuple (with the result of f applied to the "preceding" tuple, appended with the preceding tuple) and apply the function defined by h to it. We define then the function p_f by:
  $$p_f(n,m,g,h,f_g,f_h) \equiv \lambda(t.natrec(apply(f_g, tl(t)),\; u.v.apply(f_h, \langle v, chgfst(u,t)\rangle),\; hd(t))) \in T(succ(n)) \rightarrow T(m) \;\; [n, m \in N,\; g \in TPR(n,m),\; h \in TPR(succ(m+n), m),\; f_g \in T(n) \rightarrow T(m),\; f_h \in T(succ(m+n)) \rightarrow T(m)].$$
- For the appending of g and h, we use the constant $\langle\, ,\, \rangle$ to define the function a_f, which appends the results of the application of the functions defined by g and h to a tuple:
  $$a_f(n,m_1,m_2,g,h,f_g,f_h) \equiv \lambda(t.\langle apply(f_g,t), apply(f_h,t)\rangle) \in T(n) \rightarrow T(m_1+m_2) \;\; [n, m_1, m_2 \in N,\; g \in TPR(n,m_1),\; h \in TPR(n,m_2),\; f_g \in T(n) \rightarrow T(m_1),\; f_h \in T(n) \rightarrow T(m_2)].$$

Now, under the assumptions $n, m \in N$, $f \in TPR(n,m)$, we can define the constant:

$$TPR_{fun}(n,m,f) \equiv TPRe(z_f,\; s_f,\; r_f,\; c_f,\; p_f,\; a_f, n, m, f) \in T(n) \rightarrow T(m),$$

and then the result of the application of f in $TPR(n,m)$ to a tuple t in $T(n)$ is just:

$$Apply_n^m(f,t) \equiv apply(TPR_{fun}(n,m,f),\, t) \ \epsilon\, T(m).$$

Properties of Id
Assume A Set, $P(x)$ Set $[x \,\epsilon\, A]$, B Set, $f(x) \,\epsilon\, B$ $[x \,\epsilon\, A]$, $a, b, c \,\epsilon\, A$. Then, we can prove:

$a =_A a$ (reflexivity),

$a =_A b \supset b =_A a$ (symmetry),

$(a =_A b \;\&\; b =_A c) \supset a =_A c$ (transitivity),

$(a =_A b \;\&\; P(a)) \supset P(b)$ (substitutivity),

$a =_A b \supset f(a) =_B f(b)$ (congruence).

Properties of addition
The addition of two elements of the set N is defined by the expression:

$$x{+}y \equiv natrec(y,\, u.v.succ(v),\, x),$$

from which we can prove the following propositions:

$0{+}x =_N x \;\; [x \,\epsilon\, N]$,

$succ(y){+}x =_N succ(y{+}x) \;\; [x, y \,\epsilon\, N]$,

$x{+}y =_N y{+}x \;\; [x, y \,\epsilon\, N]$,

$x{+}(y{+}z) =_N (x{+}y){+}z \;\; [x, y, z \,\epsilon\, N]$.

Properties of order
The correctness of the definition of $<$ can be shown by proving the following proposition:

$$x < y \;\Leftrightarrow\; \exists k \epsilon\, N.\, succ(k){+}x =_N y \;\; [x, y \,\epsilon\, N].$$

Assume $x, y, z \,\epsilon\, N$, then, using this proposition, we can prove the following properties of $<$ and $\leq$:

$x < succ(x)$,

$(x < y \;\&\; y < z) \supset x < z$,

$(x < y \;\&\; y < z) \supset succ(x) < z$,

$x < (x{+}succ(y))$,

$x < y \supset (x{+}z < y{+}z)$,

$\neg\,(x < x)$,

$x \leq y \Leftrightarrow \exists k \epsilon\, N.\, k{+}x =_N y$,

$0 \leq x$,

$$(x \leq y \;\&\; y \leq z) \supset x \leq z,$$

$$x < y \supset succ(x) \leq y,$$

$$x \leq y \supset (x{+}z \leq y{+}z).$$

Properties of $\| \; \|$

The following properties of $\| \; \|$ are proved using the above properties of $=_A$, $<$ and $\leq$:

$$\|t\| =_N 0 \;\; [t \,\epsilon\, T(0)],$$

$$\|t\| =_N hd(t){+}\|tl(t)\| \;\; [n \,\epsilon\, N,\; t \,\epsilon\, T(succ(n))],$$

$$\|[k]\| =_N k \;\; [k \,\epsilon\, N],$$

$$\|chgfst(a,t)\| =_N a{+}\|tl(t)\| \;\; [a, n \,\epsilon\, N,\; t \,\epsilon\, T(succ(n))],$$

$$\|\langle t_1, t_2\rangle\| =_N \|t_1\|{+}\|t_2\| \;\; [n_1, n_2 \,\epsilon\, N,\; t_1 \,\epsilon\, T(n_1),\; t_2 \,\epsilon\, T(n_2)],$$

$$proj^i_n(t) \leq \|t\| \;\; [n, i \,\epsilon\, N,\; t \,\epsilon\, T(succ(i{+}n))],$$

$$\|t_2\| \leq \|\langle t_1, t_2\rangle\| \;\; [n_1, n_2 \,\epsilon\, N,\; t_1 \,\epsilon\, T(n_1),\; t_2 \,\epsilon\, T(n_2)].$$

Properties of *Apply*

From the definition of *Apply* we get immediately the following equalities (we omit the variables n and m in $Apply^m_n(f,t)$ in order to improve readability):

$$Apply(Z, t) = [0] \;\epsilon\; T(1) \;\; [t \,\epsilon\, T(0)],$$

$$Apply(S, [x]) = [succ(x)] \;\epsilon\; T(1) \;\; [x \,\epsilon\, N],$$

$$Apply(\pi(n,i), t) = [proj^i_n(t)] \;\epsilon\; T(1) \;\; [n, i \,\epsilon\, N,\; t \,\epsilon\, T(succ(i{+}n))],$$

$$Apply(comp(h,g), t) = Apply(h, Apply(g,t)) \;\epsilon\; T(m)$$
$$[n, k, m \,\epsilon\, N,\; h \,\epsilon\, TPR(k,m),\; g \,\epsilon\, TPR(n,k),\; t \,\epsilon\, T(n)],$$

$$Apply(app(g,h), t) = \langle Apply(g,t), Apply(h,t)\rangle \;\epsilon\; T(m_1{+}m_2)$$
$$[n, m_1, m_2 \,\epsilon\, N,\; g \,\epsilon\, TPR(n,m_1),\; h \,\epsilon\, TPR(n,m_2),\; t \,\epsilon\, T(n)].$$

For *pr* we can prove the following propositions:

$$hd(t) =_N 0 \supset (Apply(pr(g,h),t) =_{T(m)} Apply(g, tl(t)))$$
$$[n, m \,\epsilon\, N,\; g \,\epsilon\, TPR(n,m),\; h \,\epsilon\, TPR(succ(m{+}n), m)],$$

$$hd(t) =_N succ(x) \supset (Apply(pr(g,h),t) =_{T(m)}$$
$$Apply(h, \langle Apply(pr(g,h), chgfst(x,t)), chgfst(x,t)\rangle))$$
$$[n, m \,\epsilon\, N,\; g \,\epsilon\, TPR(n,m),\; h \,\epsilon\, TPR(succ(m{+}n), m),\; x \,\epsilon\, N].$$

3.4 The Proof

The formal proof is carried out as a direct translation of the proof presented in section 2.2, using the properties mentioned in the previous section. We first have to define Ackermann's function. We will do it by defining a family of functions Ack, such that $Ack(n) \in TPR(1,1)$ $(n \in N)$, in the same way we defined Ack_n in section 2.1, by induction on n: Ack_0 is the successor function, and to define Ack_{n+1} in terms of Ack_n, we use the primitive recursive schema, with the constant function yielding $Ack_n(1)$ in the 0 case, and $(Ack_n \circ \pi_2^1)$ in the successor case.
So, we first define the constant functions $K_k \in TPR(0,1)$ by induction on k:

$$K_k \equiv natrec(Z,\, u.v.comp(S,v),\, k) \in TPR(0,1) \quad [k \in N],$$

from which we can prove

$$Apply(K_k, t) =_{T(1)} [k] \quad [k \in N,\, t \in T(0)].$$

And now, we can define:

$$Ack(n) \equiv natrec(S,\, u.v.pr(K_{hd(Apply(v,\,[1]))}, comp(v, \pi(1,0))),\, n) \in TPR(1,1) \quad [n \in N]$$

$$A(x,y) \equiv hd(Apply(Ack(x), y)) \in N \quad [x, y \in N].$$

With this last definition, we can prove the propositions corresponding to properties **A1** to **A12**. The proofs of these properties are generally by induction over natural numbers, and have the form of long chains of equalities and inequalities; so, the properties of $<$, $\leq$, $+$ and $=_A$ that we listed above are enough to carry them out.

Then, to prove that for any t-primitive recursive function $f : N^n \to N^m$, there exists a constant k_f such that, for all $t \in N^n$, $\|f(t)\| < A(k_f, \|t\|)$, we define a new constant:

$$C_{th}(n,m,f) \equiv \exists k \in N.(\forall t \in T(n).\, \|Apply(f,t)\| < A(k, \|t\|))$$

which is a set (proposition) under the assumptions $n, m \in N$, $f \in TPR(n,m)$. To prove that $C_{th}(n,m,f)$ is true for all elements in the sets TPR, we will use TPR elimination. This is just the same we did in the proof in section 2.2, so the translation of the proof is straightforward, using the properties we stated in the previous section and the corresponding translations of **A1** to **A12**.

To complete the proof, we have to show that A is not in $TPR(2,1)$. This is done by proving the following proposition:

$$\neg(\exists f \in TPR(2,1).(\forall x, y \in N.\, A(x,y) =_N \|Apply(f, [x,y])\|))$$

(where $[x,y] \equiv cons(x, [y])$). This is proved following the same steps as in section 2.1, for which we use asymmetry of $<$.

3.5 Editing the Proof

We used ALF [ACN90] both to represent Martin-Löf's set theory and to edit and check the proof. Using ALF's type system, a version of Martin-Löf's (monomorphic) set theory was developed[2]. To each set corresponds an ALF type; the membership relation between elements and sets is identified with the membership relation between objects and types; equal sets correspond to

[2] We followed the lines of the representation of Martin-Löf's monomorphic set theory in Martin-Löf's logical framework [NPS90].

equal types; and equal elements in a set to equal objects in the corresponding type. Propositions (interpreted as sets) are then also represented by types, and proof objects become explicit as objects of the corresponding type. The judgements in the set theory are translated to ALF judgements according to these rules. To the hypothetical judgements, function types are assigned: for instance, the judgement $a \epsilon A$ $[x \epsilon B]$ is directly translated to $a{:}A\,[x{:}B]$, from which an object in a function type can be obtained: $x.a{:}(x{:}B)A$. The rules are translated into constants belonging to function types, from the types of the premisses to the type of the conclusion. So, new sets are defined by constants corresponding to the rules defining them in the set theory.

In order to structure the theory the definitions were grouped in *logics*, each one (generally) depending on previously defined ones. For instance, to define the set N of natural numbers, we define the logic N, which has the following constants:

N: *Type*,

0: N,

succ: $(n{:}N)N$,

natrec: $(C{:}(n{:}N)\ \mathit{Type})(d{:}C(0))(e{:}(u{:}N)(v{:}C(u))C(\mathit{succ}(u)))(n{:}N)C(n)$.

The logic T of tuples, based on the logic N, has the following declarations:

T: $(n{:}N)\,\mathit{Type}$,

nil: $T(0)$,

cons: $(n{:}N)(a{:}N)(t{:}T(n))T(\mathit{succ}(n))$,

$$
\begin{aligned}
\mathit{Telim}\colon\ & (C{:}(n{:}N)(t{:}T(n))\ \mathit{Type}) \\
& (d{:}C(0, \mathit{nil})) \\
& (e{:}(n{:}N)(a{:}N)(t{:}T(n))(u{:}C(n,t))\,C(\mathit{succ}(n), \mathit{cons}(n,a,t))) \\
& (n{:}N)\ (t{:}T(n)) \\
& C(n,t).
\end{aligned}
$$

The operations and properties mentioned in the previous section were respectively defined and proved using ALF's proof editor. All the theorems were proved completely: that is, the only constants which are not abbreviations are the ones corresponding to formation, introduction and elimination rules of the sets presented. The complete proof can be found in [Sza91].

4 About Design Decisions

In this section we will refer to the development of the proof we presented. We will mention some problems we had to face when formalizing the proof, and show how we solved (some of) them. We will also show alternative implementations of the sets we defined in section 3.2.

4.1 First Attempt

The first proof we tried to formalize was the one mentioned in section 2.1 for primitive recursive functions. When trying to formalize it in Martin-Löf's set theory, we had some problems to define the primitive recursive functions. We defined an inductive family of sets PR over N, in such a way that an element in $PR(n)$ would represent a primitive recursive function in $N^n \to N$. The introduction rules had to reflect Definition 1 in section 2.1, so we introduced one canonical constant for each way of building a primitive recursive function. But, as mentioned in section 2.2, in the case of definition of a function $f : N^n \to N$ by composition of $h : N^m \to N$ and $g_1, ..., g_m : N^n \to N$, we need to express that we have m functions $g_1, ..., g_m$, where m is a variable in the rule. One way to do it was by having a family of functions $g(i)$ over N, for $i \leq m$. Then, the introduction rule for the composition case takes a function $h \in PR(m)$, and a functional expression g, such that $g(i) \in PR(n)$ whenever $i \leq m$[3]. This seemed to be a good choice, but when proving equalities (the *Id* proposition represents an *intensional* equality), we could not prove that two functional expressions were equal just by knowing that they yielded equal values when applied to equal elements in the domain.

This problem became crucial when proving that the application of the constant function K_k to a tuple in $T(0)$ yields k, which is used to prove that $A(x+1, y+1) = A(x, A(x+1, y))$, and in the induction step for the composition case of the proof that for any t-primitive recursive function $f : N^n \to N^m$, there exists a constant k_f such that for all $x_1, ..., x_n \in N$, $f(x_1, ..., x_n) < A(k_f, \sum_{i=1}^n x_i)$.

The extension proposed in section 2.2 seems to be an elegant way to get rid of these problems: we internalize the notion of tuples by considering functions in $N^n \to N^m$, and with this, the composition operator becomes uniform in the sense that it always takes two functions to yield a new one. The choice of having an appending operator instead of a tupling one is for exactly the same reason.

4.2 Inductive Definitions

In section 3.3 we gave alternative definitions of some operations on tuples using higher order functions, in order to be able to prove equalities involving these operations in an easier way. When defining families of sets, we can consider some of the sets appearing in the formation and introduction rules as parameters, in which case we obtain a set (or a family of sets) for each element in the parameter set. As a consequence, we get one elimination rule for each set (family) we defined, that is, one for each element in the parameter set of the definition.

For instance, if in the definition of the sets $T(n)$ in section 3.2 we consider n to be a parameter, we obtain one set $T(n)$ for each $n \in N$. This is done by defining independently a set Tz which is the singleton *nil*, and a set forming operator Ts taking a set as argument and such that elements in $Ts(A)$ (where A is a set) are of the form $cons(a, t)$, for $a \in N$ and $t \in A$. Then, using universes we can define:

$$T(n) \equiv Set(natrec(\widehat{Tz},\ u.v.\widehat{Ts(v)},\ n))$$

which is a set whenever $n \in N$.
With these definitions we can define partial functions over elements of these sets, like *hd* and *tl* (see [Sza91]).

The order on natural numbers was also defined as a binary relation over N. If we consider the first index of the family as a parameter, we obtain, for each $n \in N$, a family of sets $Less(n)$ over N such that an element in $Less(n)(m)$ is a proof that $n < m$. From these, we obtain a

[3] In order to avoid having an introduction rule depending on a a proposition to be true, we defined a family of sets Le over N, in such a way that the elements in $Le(n)$ were "copies" of the natural numbers which are less than or equal to n (i.e, define inductively $Le(n)$ by $zz \in Le(n)$ for all $n \in N$, and if $k \in Le(n)$, then $ss(k) \in Le(succ(n))$). The introduction rule for the composition case took then $h \in PR(m)$ and $g(i) \in PR(n)\ \ [i \in Le(m)]$.

definition of $n < m$ by induction on n, with which some properties of $<$, like transitivity, are much easier to prove (remember that to prove the transitivity of $<$ we had to prove first that $n < m$ was (logically) equivalent to $\exists k \epsilon N.\, succ(k) + n =_N m$, and then use this property to prove the transitivity).

It is not clear yet whether it is convenient to consider indexes as parameters, and the relation between the rules with and without parameters is not known in the general case either.

In the case of $T(n)$, if we consider n to be a parameter, we can prove some "closure" properties like $\Pi t \epsilon T(0).\; t =_{T(0)} nil$ in a very direct form, from which some properties needed in the proof, like $\Pi t \epsilon T(0).\; \|t\| =_N 0$ follow without having to use higher order functions in the definition of $\| \; \|$ (we can prove the same property with the original definition of the family T using universes[4]).

In the case of $<$, the two ways of defining the relation seem to be useful when proving properties of it.

4.3 The Abstract Specification Approach

As it is common in programming, it is possible in ALF to work with abstract sets and operations rather than with their actual inductive and recursive definitions. Thus, we may specify abstract structures and operations, which consist of a collection of propositions implicitly characterizing a number of sets and operations on them (such specification is usually given by a set of equations).

When writing the proof, we used this approach in order to work independently of their definitions. The usual definition of tuples, for instance, is done by requiring (a family of) sets $T(n)$ for $n \epsilon N$, having the following operations:

$nil \epsilon T(0)$,

$cons$: $N \times T(n) \rightarrow T(succ(n))$, for all $n \epsilon N$,

hd: $T(succ(n)) \rightarrow N$, for all $n \epsilon N$,

tl: $T(succ(n)) \rightarrow T(n)$, for all $n \epsilon N$,

such that the following equalities hold:

$hd(cons(a,t)) = a$, for all $a \epsilon N$, $t \epsilon T(n)$ $(n \epsilon N)$,

$tl(cons(a,t)) = t$, for all $a \epsilon N$, $t \epsilon T(n)$ $(n \epsilon N)$.

Using hd and tl, we can define all the operations on tuples of section 3.3 and all the properties of these operations can be proved from these definitions using only the two equalities stated above.

We have three different implementations of the tuples, the two already presented, plus a definition using universes to define the set $T(n)$ as $N \times N \times \ldots \times N$, n times[5]. We defined hd and tl for each of these three definitions and proved that the equalities stated above hold for each implementation.

For the order on the natural numbers, we used the same approach, and found the set of properties of this relation we used. We then implemented it in three different ways, the two already presented, plus the definition of $n < m$ as the proposition $\exists k \epsilon N.\, succ(k) + n =_N m$.¡

[4]this was pointed out to me by T. Altenkirch.

[5]The definition is by induction on n, and for the case $n = 0$, the set with only one element ($\top$) was chosen.

References

[Ack28] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.

[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In G. Huet and G. Plotkin, editors, *Informal Proceedings of the First Workshop on Logical Frameworks*, pages 39–42. Esprit Basic Research Action 3245, May 1990.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In *Proceedings of COLOG-88*, number 417 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[Dyb90] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Informal Proceedings of the First Workshop on Logical Frameworks*, pages 213–230. Esprit Basic Research Action 3245, May 1990. To appear in G. Huet and G. Plotkin, editors, *Logical Frameworks.*

[Her65] H. Hermes. *Enumerability, Decidability, Computability.* Springer-Verlag, Berlin, 1965.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pét35] R. Péter. Konstruktion nichtrekursiver Funktionen. *Mathematische Annalen*, 111:42–60, 1935.

[Sza91] Nora Szasz. A Machine Checked Proof that Ackermann's Function is not Primitive Recursive. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, June 1991.