PROCEEDINGS OF THE 1992

WORKSHOP ON

TYPES FOR PROOFS AND PROGRAMS

Båstad

June 1992

Eds Bengt Nordström, Kent Petersson and Gordon Plotkin

Contents

Types *and* *Programs*

Foreword

This document is the preliminary proceedings of the 1992 Workshop on Types for Proofs and Programs held at Hotel Riviera in Båstad, Sweden, from the 8th to the 12th of June 1992. The workshop was organized by Peter Dybjer, Bengt Nordström, Kent Petersson and Jan Smith from Göteborg together with Rod Burstall and Claire Jones from Edinburgh. The workshop was attended by 55 people.

Local arrangements were looked after by Marie Larsson et al and Lennart Augustsson and Per Lundgren took care of the computer equipment.

These preliminary proceedings have been collected from LaTeX sources, using electronic mail and spliced together. They comprise 18 of the 31 papers presented at the meeting (11 abstracts are also included).

This document may be obtained by anonymous ftp from Chalmers University of Technology (compressed file `pub/baastad/proc.ps.Z` on `animal.cs.chalmers.se` login name “anonymous” and your ordinary name as password).

Workshop Programme: Types for Proofs and Programs

Sunday, June 7th

8.00pm	Dinner

Monday, June 8th

9.00–10.00am	*Per Martin-Löf* Does denotational semantics have any role to play in type theory?
10.00–10.30am	*Peter Aczel* A LEGO implementation of constructive set theory
10.30–11.15am	Coffee Break
11.15–12.00am	*Zhaohui Luo* Compositional understanding of type theory
12.00–12.25pm	*Veronica Gaspes and Jan Smith* Machine checked normalization proofs for typed combinator calculi
12.25–12.50pm	*Veronica Gaspes* Formal proofs of combinatorial completeness
12.50–3.00pm	Lunch
3.00–3.45pm	*Vincent Danos and Laurent Regnier* Virtual reduction
3.45–4.10pm	*Christine Paulin-Mohring* Representation of specifications and programs involving streams in the Calculus of Inductive Constructions
4.10–4.35pm	*Lena Magnusson* The new implementation of ALF
4.35–5.15pm	Coffee Break
5.15–6.00pm	Demo Session
8.00pm	Dinner

Tuesday, June 9th

9.00–9.45am	*Thierry Coquand* Contexts, substitutions and pattern-matching
9.45–10.15am	*Joelle Despeyroux* Natural semantics in Coq, first experiments
10.15–10.30am	*Piotr Rudnicki* Mizar
10.30–11.15am	Coffee Break
11.15–11.40am	*Leen Helmink* Girard's paradox in lambda-U
11.40–12.50pm	DEMO SESSION
12.50–3.00pm	Lunch
3.00–3.45pm	*Benjamin Werner* Strong normalization of lambda-calculi with types defined by recursion
3.45–4.30pm	*James McKinna* Deliverables: an approach to program development
4.30–5.15pm	Coffee break
5.15–6.00pm	PANEL SESSION
6.00–7.30pm	Site leaders meeting
7.30pm	Dinner
9.00pm	Concert (in Båstad Church)

Wednesday, June 10th

9.00–9.45am	*Michel Parigot* $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction
9.45–10.30am	*Hugo Herbelin* Computable interpretation of cross-cuts procedure
10.30–11.15am	Coffee Break
11.15–12.00am	*Stefano Berardi* Game theory and program extraction
12.00–12.25pm	*Franco Barbanera* Continuations and simple types: A strong normalisation result
12.25–12.50pm	*Thomas Streicher* Truly intensional models for type theory arising from modified realizability
12.50–2.45pm	Lunch
2.45pm	FREE AFTERNOON Bus leaves for walks (various distances) to Hovs hallar
7.00pm	Dinner in Torekov (bus from Hovs hallar at 6.00pm)

Thursday, June 11th

9.00–9.45am	*Peter Aczel* Schematic consequence
9.45–10.30am	*Randy Pollack* Type checking in Pure Type Systems
10.30–11.15am	Coffee Break
11.15–12.00pm	*Bert Jutting* Typing in Pure Type Systems
12.00–12.45pm	*Alex Simpson* Kripke semantics for a logical framework
12.45–3.00pm	Lunch
3.00–3.25pm	*Catarina Coquand* The proof of normalisation for simply typed lambda calculus written in ALF
3.25–3.50pm	*Daniel Fridlender* Formalizing proofs of properties of well-quasi-orders
3.50–4.35pm	*Frank Pfenning* Teaching theory of programming languages using a logical framework: an experience report
4.35–5.15pm	Coffee Break
5.15–6.00pm	*Christophe Raffalli* Fixed point and type systems
8.00pm	Dinner

Friday, June 12th

9.00–10.00am	*Jim Lipton* Logic programming and propositions as types: A realizability interpretation of declarative programs
10.00–10.30am	*David Pym and Gordon Plotkin* A relevant analysis of natural deduction
10.30–11.15am	Coffee Break
11.15–12.00am	*Phillipe Audebaud* An extension of the calculus of constructions using fixpoints
12.00–12.45pm	*Peter Dybjer* Simultaneous inductive-recursive definitions in type theory
12.45–3.00pm	Lunch

List of Participants

INRIA, Rocquencourt

Samuel Boutin	boutin@margaux.inria.fr
Hugo Herbelin	herbelin@margaux.inria.fr
Gérard Huet	huet@margaux.inria.fr
Chet Murthy	murthy@margaux.inria.fr
Benjamin Werner	werner@margaux.inria.fr

Lyon

Catherine Parent	parent@lip.ens-lyon.fr
Christine Paulin	cpaulin@lip.ens-lyon.fr

Eindhoven

Leen Helmink	helmink@prl.philips.nl
Erik Poll	erik@win.tue.nl

Edinburgh/Oxford

Rod Burstall	rb@dcs.ed.ac.uk
Philippa Gardner	pag@dcs.ed.ac.uk
Claire Jones	ccmj@dcs.ed.ac.uk
Zhaohui Luo	zl@dcs.ed.ac.uk
Savi Maharaj	svm@dcs.ed.ac.uk
James McKinna	jhm@dcs.ed.ac.uk
Randy Pollack	rap@dcs.ed.ac.uk
David Pym	dpym@dcs.ed.ac.uk
Alex Simpson	als@dcs.ed.ac.uk

Manchester

Peter Aczel	petera@cs.man.ac.uk

Nijmegen

Henk Barendregt	henk@cs.kun.nl
Bert van Benthem Jutting	
Mark Ruys	markr@cs.kun.nl

Paris 7

Vincent Danos	danos@logique.jussieu.fr
Jean-Louis Krivine	krivine@logique.jussieu.fr
Pascal Manoury	eleph@logique.jussieu.fr
Patrick Meche	meche@logique.jussieu.fr
Michel Parigot	parigot@logique.jussieu.fr
Christophe Raffalli	raffalli@logique.jussieu.fr
Laurent Regnier	regnier@logique.jussieu.fr
Marianne Simonot	simonot@logique.jussieu.fr

Sophia-Antipolis

Joëlle Despeyroux	jd@sophia.inria.fr

Torino

Stefano Berardi	stefano@di.unito.it
Franco Barbanera	barba@di.unito.it
Pietro Di Gianantonio	digianant@uduniv.cineca.it

Göteborg

Lennart Augustsson	augustss@cs.chalmers.se
Jan Cederqvist	ceder@cs.chalmers.se
Catarina Coquand	catarina@cs.chalmers.se
Thierry Coquand	coquand@cs.chalmers.se
Peter Dybjer	peterd@cs.chalmers.se
Veronica Gaspes	vero@cs.chalmers.se
Michael Hedberg	hedberg@cs.chalmers.se
Daniel Friedlender	frito@cs.chalmers.se
Lena Magnusson	lena@cs.chalmers.se
Bengt Nordström	bengt@cs.chalmers.se
Kent Petersson	kentp@cs.chalmers.se
Jan Smith	smith@cs.chalmers.se
Nora Szasz	nora@cs.chalmers.se
Alvaro Tasistro	tato@cs.chalmers.se

Others

Jean-Paul Audebaud	paudebau@lip.ens-lyon.fr
Eduardo Gimenez	gimenez@incouy.edu.uy
Jim Lipton	lipton@saul.cis.upenn.edu
Per Martin-Löf	Barnhusgatan 4, 111 23 Stockholm, Sweden
Frank Pfenning	Frank.Pfenning@cs.cmu.edu
Piotr Rudnicki	piotr@cs.ualberta.ca
Anne Salvesen	abs@ifi.uio.no
Thomas Streicher	streicher@informatik.uni-muenchen.de

Demonstrations

In the demo-sessions the following systems were shown:

- ALF-2 Lennart Augustsson and Thierry Coquand,
- Coq Catherine Parent and Benjamin Werner,
- Elf Frank Pfenning.
- Lego Randy Pollack.

Schematic Consequence*

Peter Aczel
Manchester University

August 1992

Abstract

The aim of this note is first to set up some general theory for discussing different aspects of the notion of a logic and then to draw attention to the schematic aspects of logic and suggest a way of capturing this aspect without making any commitment to the kind of syntax a logic should have.

Introduction

Nowadays we are well aware that there are many different logics. There are computer systems which are meant to be used to implement many logics. But there is no generally accepted account of what a logic is. Perhaps this is as it should be. We need imprecision in our vocabulary to mirror the flexible imprecision of our thinking. There are a number of related phrases that seem to have a similar imprecision; e.g. formal system, language, axiom system, theory, deductive system, logical system, etc... These are sometimes given technical meanings, often without adequate consideration of the informal notions.

When a logic has been implemented in a computer system the logic has been represented in the logical framework that the computer system uses. The logical framework will involve a particular approach to syntax, which may differ from the approach to syntax taken when the logic was first presented. This means that in order to represent the logic, as first presented, in the framework a certain amount of coding may be needed and the question will arise whether the logic as first presented is indeed the logic that has been represented in the framework. To make this question precise it is necessary to have a notion of a logic that abstracts away from particular approaches to syntactic presentation.

So we want to have a formal abstract definition of the notion "a logic". It is probable that there is no single notion to be captured but rather several related notions hiding under the single phrase. Nevertheless it would seem to be very worthwhile to try to analyse the situation and come up with some technical definitions that can be used to capture the various facets of the notion(s) of a logic. Of course there has been previous work on this topic. An important ingredient in all recent work is the notion of a consequence relation. This is essentially due to Tarski who developed a fairly general theory of consequence operators and deductive systems in the 1920s. Dana Scott has more recently focused on the notion of a consequence relation.

*This paper is incomplete and reports on work in progress. An earlier version was called *The Notion of a Logic* and was based on a talk given at the workshop on "Deductive Systems" at Oberwolfach in May 1991. My attention was drawn to the adjoint situations in the topic by a question after my talk. This work was partially supported by the Esprit Basic Research Action 'Logical Frameworks' and an SERC research Fellowship.

Another important development was Barwise's abstract model theory ([Barwise 1974], which gave a model theoretic approach to a general class of logics, those based on classical first order semantic structures. A further stage of abstraction gives rise to Goguen and Burstall's notion of an institution, ([Goguen and Burstall, 1990]). This level of abstraction seems to be needed to capture the variety of logics that seem to be of relevence in computer science and linguistics. The notion of an institution only focuses on the semantic aspect of logical consequence. Meseguer, in [Meseguer, 1989], has given a definition of a logic which uses an institution to capture the semantic aspect but also uses an entailment system to capture the operational aspect. An entailment system is roughly a system of consequence relations indexed by signatures. In Meseguer's notion of a logic the consequence relations of the entailment system are required to be sound with respect to the semantic consequence relations of the institution. The notion of an entailment system has also been used in [Harper, Sannella and Tarlecki, 1990?], where they called it a logical system or logic.

At this point we should also mention the Edinburgh Logical Framework ([Harper, Honsell and Plotkin, 1987]) and other systems playing similar roles in connection with computer systems for implementing logics. These logical frameworks provide an environment in which the operational, non-semantic aspects of a logic can be specified, provided that a syntax for the logic is acceptable to the framework or can be made acceptable without essentially changing the logic. One approach to the notion of a logic might be to choose a logical framework and require that the operational part of a logic must have a representation in the logical framework.

What are the key issues in connection with the notion(s) of a logic? We list some of them.

1. Logic is concerned with what follows from what; i.e. with logical consequence.

2. Logic is generally applicable; i.e. a logic should have a range of possible applications so that if A follows logically from B in a logic then in every application (or interpretation or context) where A is true B is true.

3. Logic is schematic; e.g. the rule of modus ponens, that B follows logically from A and $A \to B$, is not expressed in the language of any particular application of the logic, but in a language of schematic expressions involving parameters which can be instantiated in any application language.

In the next section below we give some definitions relevent to the first issue. We focus on finitary traditional consequence. More general notions are of interest. See for example [Avron, 1991]. But the more restricted and familiar notion of a consequence relation will be good enough for our present purposes. We also introduce several other notions which are relevent in connection with the formation of consequence relations either operationally from an axiomatic system or semantically from a semantic system. Both methods of formation can be seen to use the notion of a truth system. We also define a hybrid notion incorporating a semantic system with an axiomatic system that is sound.

We address the second issue above by indexing the above notions in section 3. Finally we address the third and less familiar issue in section 4. There is a good deal more to be said on these topics and we hope to produce an expanded version of this paper. In section 2 we make a slight detour from the main topic to consider the connections with Scott's notion of an information system and domains.

1 Finitary Traditional Consequence

We describe five categories relevent to an abstract theory of finitary traditional consequence and study the relationships between them.

Axiomatic systems

An *axiomatic system* (A, Φ) consists of a set A and a set $\Phi \subseteq (finA) \times A$, where $finA$ is the set of finite subsets of A. Axiomatic systems form a category AS. A map $\pi : (A, \Phi) \to (A', \Phi')$ of AS is a function $\pi : A \to A'$ such that

$$(\Gamma, \theta) \in \Phi \;\Rightarrow\; (\pi\Gamma, \pi\theta) \in \Phi$$

where $\pi\Gamma = \{\pi\gamma \mid \gamma \in \Gamma\}$ if $\Gamma \in FinA$.

We think of an axiomatic system as specifying a notion of proof whose inference steps have the form

$$\frac{\gamma_1 \cdots \gamma_n}{\theta}$$

where $(\{\gamma_1, \ldots, \gamma_n\}, \theta) \in \Phi$. Here of course $\gamma_1, \ldots, \gamma_n$ are the premises of the inference step and θ is the conclusion.

Consequence Relations

A *consequence relation* $(A, \vdash)$ is an axiomatic system satisfying the following.

$$\begin{array}{rcl}
\textbf{reflexivity} & : & \{\theta\} \vdash \theta \\
\textbf{monotonicity} & : & \Gamma \vdash \theta \;\Rightarrow\; \Gamma \cup \{\varphi\} \vdash \theta \\
\textbf{transitivity} & : & \Gamma \vdash \varphi \text{ and } \Gamma \cup \{\varphi\} \vdash \theta \;\Rightarrow\; \Gamma \vdash \theta
\end{array}$$

Consequence relations form a full subcategory CR of AS.

Proposition 1 *The inclusion functor $CR \hookrightarrow AS$ has a left adjoint $AS \to CR$ which associates with each axiomatic system (A, Φ) the consequence relation* $(A, \vdash_\Phi)$ generated *by (A, Φ); i.e. $\vdash_\Phi$ is the least relation including Φ such that $(A, \vdash_\Phi)$ is a consequence relation, or alternatively*

$$\Gamma \vdash_\Phi \theta \iff \theta \textit{ is in the smallest } \Phi\textit{-closed set including } \Gamma,$$

where $X \subseteq A$ is Φ-closed *if whenever $(\Gamma, \theta) \in \Phi$*

$$\Gamma \subseteq X \;\Rightarrow\; \theta \in X.$$

Truth Systems

A *truth system* (A, M) consists of a set A and a set $M \subseteq powA$, where $powA$ is the set of all subsets of A. Truth systems form a category TS, where a map $\pi : (A, M) \to (A', M')$ of TS is a function $\pi : A \to A'$ such that

$$X \in M' \;\Rightarrow\; \pi^{-1}X \in M.$$

With each axiomatic system (A, Φ) we may associate the truth system $(A, cl(\Phi))$ where $cl(\Phi)$ is the set of Φ-closed subsets of A.

Proposition 2 *$(A, \Phi) \mapsto (A, cl(\Phi))$ can be made into a functor $AS \to TS$ which factorises $AS \to CR \to TS$. Moreover $CR \to TS$ has a right adjoint $TS \to CR$ which associates with each truth system (A, M) the consequence relation $(A, \models_M)$ where*

$$\Gamma \models_M \theta \iff (\forall X \in M)[\,\Gamma \subseteq X \;\Rightarrow\; \theta \in X\,].$$

Closure Systems

A *closure system* is a truth system (A, M) such that

1. $\bigcap \mathcal{X} \in M$ for all $\mathcal{X} \subseteq M$ (where we take $\bigcap \mathcal{X} = A$ when $\mathcal{X} = \emptyset$),
2. $\bigcup \mathcal{X} \in M$ for all directed $\mathcal{X} \subseteq M$.

Let CS be the full subcategory of TS whose objects are the closure systems.

Proposition 3 *The adjoint functors $CR \to TS$ and $TS \to CR$ have factorisations $CR \to CS \hookrightarrow TS$ and $TS \to CS \to CR$ which are also adjoints. Moreover the functors $CR \to CS$ and $CS \to CR$ are inverses of each other, so that $CR \cong CS$.*

Semantical Systems

A *semantical system* $(A, Mod, \models)$ consists of a set A, a class Mod and a class relation $\models$ that is a subclass of $Mod \times A$. Semantical systems form a category SS, where a map $(\pi_1, \pi_2) : (A, Mod, \models) \to (A', Mod', \models')$ of SS consists of a function $\pi_1 : A \to A'$ and a function $\pi_2 : Mod' \to Mod$ such that for $m' \in Mod', \theta \in A$

$$(\pi_2 m') \models \theta \iff m' \models' (\pi_1 \theta).$$

Each truth system (A, M) determines a semantical system $(A, M, \ni_M)$ where $m \ni_M \theta \iff \theta \in m \in M$. This gives a functor $TS \to SS$.

Proposition 4 *The functor $TS \to SS$ has a right adjoint $SS \to TS$ which associates with each semantical system $(A, Mod, \models)$ the truth system*

$$(A, \{th(m) \mid m \in Mod\})$$

where, for each $m \in Mod$,

$$th(m) = \{\theta \in A \mid m \models \theta\}.$$

Note that the composition $SS \to TS \to CR$ gives a functor $SS \to CR$ that sends each semantical system $(A, Mod, \models)$ to the consequence relation $(A, \models\!\!\equiv)$ where

$$\Gamma \models\!\!\equiv \theta \iff (\forall m \in Mod)[\,(\forall \gamma \in \Gamma)(m \models \gamma) \;\Rightarrow\; m \models \theta\,].$$

Axiomatic system with semantics

We now describe a hybrid notion. An *axiomatic system with semantics* $(A, \Phi, Mod, \models)$ consists of an axiomatic system (A, Φ) and a semantical system $(A, Mod, \models)$ such that

$$soundness: \quad \Gamma \vdash_\Phi \theta \;\Rightarrow\; \Gamma \models\!\!\equiv \theta.$$

The axiomatic system with semantics is *complete* if the converse holds; i.e.

$$\Gamma \models\!\!\equiv \theta \;\Rightarrow\; \Gamma \vdash_\Phi \theta.$$

Axiomatic systems with semantics form a category ASS in the obvious way.

Summary

We have defined the four categories AS, CR, TS, SS with functors

$$AS \to CR \cong CS \hookrightarrow TS \to SS$$

which have right adjoints

$$AS \hookleftarrow CR \cong CS \leftarrow TS \leftarrow SS.$$

There is also the hybrid category ASS with the obvious forgetfull functors $ASS \to AS$ and $ASS \to SS$. All these categories and functors are "over the category of sets"; i.e. for each of these categories $\mathbf{C}$ there is a forgetful functor $\mathbf{C} \to Sets$ and for each of the above functors $\mathbf{C} \to \mathbf{C}'$ the diagram

$$\begin{tikzcd} \mathbf{C} \arrow[rr] \arrow[dr] & & \mathbf{C}' \arrow[dl] \\ & Sets & \end{tikzcd}$$

commutes.

Lindenbaum Algebras

A more abstract algebraic treatment of traditional consequence will work with (meet) semilattices and the Lindenbaum algebra construction. Consider the following natural way to make a (meet) semilattice $(A, \leq)$ into a consequence relation $(A, \vdash_{\leq})$, where

$$C \vdash_{\leq} a \;\Leftrightarrow\; \bigwedge C \leq a.$$

This gives rise to a functor from the category SL of semilattices to the category CR. We can characterise the Lindenbaum algebra construction as a left adjoint $CR \to SL$ to this functor that maps $(A, \vdash)$ to $(Q, \leq)$ where Q is a quotient of $finA$ with respect to the equivalence relation $\equiv$ given by

$$C_1 \equiv C_2 \;\Leftrightarrow\; \text{for all } a \in A \;\; C_1 \vdash a \text{ iff } C_2 \vdash a$$

and $\leq$ is the partial ordering on Q given by

$$[C_1] \leq [C_2] \;\Leftrightarrow\; C_1 \vdash a \text{ for all } a \in C_2.$$

2 Consequence Relations and Scott's Information Systems

This section is really a detour from the main topic. As in the previous section the categories and functors that we will consider are all "over the category of sets".

There is an interesting relationship between the notion of a consequence relation and Scott's notion of an information system, (see [Scott, 1982]). This gives rise to a relationship between closure systems and Scott domains. Essentially a Scott Information System can be viewed as a consequence relation $(A, \vdash)$ and a *consistency notion* Con for it; i.e. $Con \subseteq finA$ such that for all $C \in finA, a \in A$

1. $C \cup \{a\} \in Con \;\Rightarrow\; C \in Con,$
2. $C \not\vdash a \;\Rightarrow\; C \in Con,$

3. $C \vdash a$ and $C \in Con \;\Rightarrow\; C \cup \{a\} \in Con$.

Given such a consistency notion we can call a $\vdash$-closed set $X \subseteq A$ *consistent* if $finX \subseteq Con$. Then the set $Cl(A, \vdash, Con) = \{X \in cl(\vdash) \mid X \text{ is consistent}\}$ of 'elements' of the information system form a Scott domain, when partially ordered by the subset relation. It is a standard fact that every Scott domain is isomorphic to one of these.

We call a truth system (A, M) a *weak closure system* if it satisfies the two conditions of a closure system, except that, in condition 1, $\bigcap \mathcal{X} \in M$ need only hold for non-empty $\mathcal{X} \subseteq M$. So a closure system is simply a weak closure system (A, M) such that $A \in M$. We may form the category wCS which is the full subcategory of CS consisting of weak closure systems. Also let CRC be the category whose objects are triples $(A, \vdash, Con)$ where $(A, \vdash)$ is a consequence relation and Con is a consistency notion for it. The maps $\pi : (A, \vdash, Con) \to (A', \vdash', Con')$ in this category are the maps $\pi : (A, \vdash) \to (A', \vdash')$ in CR such that for all $X \in finA$

$$\pi X \in Con' \;\Rightarrow\; X \in Con.$$

Then we get an isomorphism

$$CRC \cong wCS$$

given by mapping $(A \vdash, Con)$ in CRC to $(A, Cl(A, \vdash, Con))$ in wCS. The inverse of this isomorphism maps (A, M) in wCS to $(A, \vdash_M, Con_M)$ in CRC, where the map $(A, M) \mapsto (A, \vdash_M)$ is given by the functor $wCS \hookrightarrow TS \to CR$ and

$$Con_M = \{C \in finA \mid C \subseteq X \text{ for some } X \in M\}.$$

Given a consequence relation $(A \vdash)$, there are two extreme ways to form a consistency notion Con for it. The maximum consistency notion is given by $Con = finA$. The minimum consistency notion is given by

$$Con = Con(A, \vdash) = \{X \in finA \mid X \not\vdash a \text{ for some } a \in A\}.$$

Of course in some cases the two consistency notions may coincide so that the consequence relation has a unique consistency notion. But in familiar cases we expect the two notions to be distinct (e.g. in a logic with negation we might expect that $\{b, \neg b\} \vdash a$ for all a so that if $X = \{b, \neg b\}$ then X is in $finA$, but not in $Con(A, \vdash)$). Can there be other consistency notions for a given consequence relation? I was initially surprised to observe that:

Proposition 5 *For any consequence relation the two extreme consistency notions are the only possible ones.*

To see why this is so let Con be a consistency notion for the consequence relation $(A, \vdash)$ which is different from $Con(A, \vdash)$. It follows that there is some $C \in Con$ such that $C \vdash a$ for all $a \in A$. We want to show that any $X \in finA$ is in Con. So let $X = \{a_1, \ldots, a_n\} \in finA$. As $C \vdash a$ we get that $C \cup \{a_1\} \in Con$. As $C \vdash a_2$ we get $C \cup \{a_1\} \vdash a_2$ so that $C \cup \{a_1, a_2\} \in Con$. Continuing in this way we eventually get that $C \cup X \in Con$. As $X \subseteq C \cup X$ it follows that $X \in Con$.

The two ways to construct consistency notions give rise to functors in a natural way, though for the second way a little care is needed. In the first case we have the functor $CR \to CRC$ that maps $(A, \vdash)$ to $(A, \vdash, finA)$.

Proposition 6 *The above functor $CR \to CRC$ is left adjoint to the forgetfull functor $CRC \to CR$ that maps $(A, \vdash, Con)$ to $(A, \vdash)$.*

For the other consistency notion construction we need to restrict to surjective functions. Let CR' be the subcategory of CR having the same objects as CR but having as maps only those maps of CR that are surjective as functions. Similarily we may define categories $CRC', CS', etc \ldots$. The above functor $CR \to CRC$ can be restricted to give a functor $CR' \to CRC'$ which is still a left adjoint to the forgetful functor on CRC'. But we also have another functor $CR' \to CRC'$ that maps $(A, \vdash)$ to $(A, \vdash, Con(A, \vdash))$. It is not possible to extend this to CR.

Proposition 7 *The latter functor $CR' \to CRC'$ is right adjoint to the forgetful functor $CRC' \to CR'$.*

So we get a sequence of three functors between CR' and CRC' where the first functor in each successive pair is left adjoint to the second.

$$\begin{array}{lllll} CR' & \to & CRC' & (A, \vdash) & \mapsto (A, \vdash, finA) \\ CRC' & \to & CR' & (A, \vdash, Con) & \mapsto (A, \vdash) \\ CR' & \to & CRC' & (A, \vdash, Con(A, \vdash)) & \mapsto (A, \vdash) \end{array}$$

The isomorphisms

$$CR \cong CS \text{ and } CRC \cong wCS$$

induce isomorphisms

$$CR' \cong CS' \text{ and } CRC' \cong wCS'$$

and hence give rise to a corresponding sequence of functors between CS' and wCS'.

$$\begin{array}{llll} CS' & \hookrightarrow & wCS' & (A, M) \mapsto (A, M) \\ wCS' & \to & CS' & (A, M) \mapsto (A, M \cup \{A\}) \\ CS' & \to & wCS' & (A, M) \mapsto (A, \overline{M}). \end{array}$$

where $\overline{M}$ is $M - \{A\}$ if this is a weak closure system and is M otherwise.

There is a close connection between the collections of weak closure systems and Scott domains. But it is too strong to assert that the category *Scott* of Scott domains and continuous maps is equivalent to the category wCS. We do get an equivalence if we restrict to categories of isomorphisms. To get a better result we can consider the contravariant functor $wCS \to Scott^{op}$ which, on objects, maps (A, M) to $(A, \subseteq_M)$, where $\subseteq_M$ is the subset relation on M, and, on maps, assigns to each $\pi : (A, M) \to (A', M')$ of wCS the inverse image continuous function $\pi^{-1} : (M, \subseteq_M) \to (M', \subseteq_{M'})$. This functor is an equivalence if *Scott* is modified so that only certain continuous functions are allowed as maps.

3 Indexed Consequence

One aspect of a logic is that there is a notion of signature, so that associated with each signature is a set of sentences that can be formed with the signature. So we postulate a category SIG over the category of sets, the functor $SIG \to Sets$ being the functor associating with each signature Σ of SIG the set $sent(\Sigma)$ of sentences of Σ.

As an example first order logic could use the category SIG whose objects consist of sets of individual constant symbols and n-place function and predicate symbols for $n > 0$. The signature maps of SIG are functions from symbols to symbols that map individual constants to individual constants, n-place function symbols to n-place function symbols and n-place predicate symbols to n-place predicate symbols. Now $sent(\Sigma)$ can be the set of first order sentences formed in the standard way using the symbols of Σ, logical symbols and individual variables.

Given SIG and $SIG \to Set$ we get *SIG-indexed notions* as functors from SIG over the category Set. For example a *SIG-indexed consequence relation* is a functor $SIG \to CR$ over Set. This is Meseguer's notion of an entailment system and also the Harper, Sanella and Tarlecki notion of a logical system.

A version of the notion of an institution is a category SIG over Set together with a SIG-indexed semantical system $SIG \to SS$ over Set.

Let CRS be the full subcategory of ASS consisting of $(A, \vdash, Mod, \models)$ in ASS where $(A, \vdash)$ is in CR. Now Meseguer's notion of a logic is a category SIG over Set together with a *SIG-indexed consequence relation with semantics* $SIG \to CRS$ over Set.

4 Schematic Consequence

We wish to capture the schematic aspects of logic. A logic is axiomatised by giving schematic axioms and rules of inference which can be instantiated to get inference steps. Instantiations may themselves be schematic so that we are interested in the notion of a derived rule of inference obtained by composing instances of the axioms and rules. (Note: We need only focus on rules, as an axiom is simply a rule with no premises.)

Substitution

We want a syntax free way of talking about instantiation. We are led to focus on substitution. We will assume that the substitutions on a set A are functions $s : A \to A$ that form a submonoid of $(A^A, id_A, \circ)$. Here A^A is the set of all functions $A \to A$, $id_A : A \to A$ is the identity function of A and if $s, s' : A \to A$ then $s \circ s' : A \to A$ is their composition. If Sub is such a submonoid then we call (A, Sub) a *concrete monoid*. The instances of $a \in A$ are the elements of A of the form sa for $s \in Sub$.

Concrete monoids form a category CM where a map $(\pi_1, \pi_2) : (A, Sub) \to (A', Sub')$ consists of a function $\pi_1 : A \to A'$ and a monoid homomorphism $\pi_2 : Sub \to Sub'$ such that for all $a \in A, s \in Sub$

$$\pi_1(sa) = (\pi_2 s)(\pi_1 a).$$

We can now give schematic versions of axiomatic systems, consequence relations etc.

Schematic Axiomatic Systems

A *schematic axiomatic system* (A, Sub, Φ) consists of a concrete monoid (A, Sub) and an axiomatic system (A, Φ). In an obvious way we get a category SAS of schematic axiomatic system.

Schematic Consequence Relations

A *schematic consequence relation* $(A, Sub, \vdash)$ is a schematic axiomatic system such that $(A, \vdash)$ is a consequence relation and for all $s \in Sub$

$$s : (A, \vdash) \rightarrow (A, \vdash) \text{ in } CR;$$

i.e.

$$\Gamma \vdash \theta \;\Rightarrow\; s\Gamma \vdash s\theta.$$

We can now let SCR be the full subcategory of SAS consisting of schematic consequence relations.

Proposition 8 *The inclusion functor* $SCR \hookrightarrow SAS$ *has a left adjoint* $SAS \rightarrow SCR$ *which associates with each schematic axiomatic system* (A, Sub, Φ) *the schematic consequence relation* $(A, Sub, \vdash_\Phi)$ generated *by* (A, Sub, Φ)*; i.e. the least relation including* Φ *such that* $(A, Sub, \vdash_\Phi)$ *is a schematic consequence relation, or alternatively*

$$\Gamma \vdash_\Phi \theta \quad \Longleftrightarrow \quad \theta \textit{ is in the smallest } Sub\Phi\textit{-closed set including } \Gamma$$

where

$$Sub\Phi = \{(s\Gamma, s\theta) \mid s \in Sub \textit{ and } (\Gamma, \theta) \in \Phi\}.$$

If $\Gamma \vdash_\Phi \theta$ *then* (Γ, θ) *is a* derived rule of Φ *(relative to* (A, Sub)*).*

Schematic Truth Systems

A *schematic truth system* (A, Sub, M) consists of a concrete monoid (A, Sub) and a truth system (A, M) such that $s : (A, M) \rightarrow (A, M)$ in TS for all $s \in Sub$; i.e.

$$X \in M \;\Rightarrow\; s^{-1}X \in M$$

for all $s \in Sub$.

STS is the category of schematic truth systems, with the obvious notion of map. With each schematic axiomatic system (A, Sub, Φ) we may associate the schematic truth system $(A, Sub, cl(Sub\Phi))$.

Proposition 9 $(A, Sub, \Phi) \mapsto (A, Sub, cl(Sub\Phi))$ *can be made into a functor* $SAS \rightarrow STS$ *which factorises* $SAS \rightarrow SCR \rightarrow STS$*. Moreover* $SCR \rightarrow STS$ *has a right adjoint* $STS \rightarrow SCR$ *which associates with each schematic truth system* (A, Sub, M) *the consequence relation* $(A, Sub, \models_M)$ *where* $\models_M$ *is obtained from* M *as in* $TS \rightarrow CR$*.*

References

[Avron, 1991] **Axiomatic Systems, Deduction and Implication**, Tech. Report 203/91, Institute of Computer Sciences, Tel-Aviv University.

[Barwise, 1974] **Axioms for Abstract Model Theory**, Annals of Mathematical Logic, 7 (1974) 221-265.

[Goguen and Burstall, 1990] **Institutions: Abstract Model Theory for Specification and Programming**, Edinburgh LFCS Report 90-106.

[Harper, Sannella and Tarlecki, 1989] **Structure and Representation in LF**, A version appeared in the proceedings of the 4th LICS meeting, 1989.

[Harper, Honsell and Plotkin, 1987] **A Framework for defining logics** In Proc. LICS 87.

[Meseguer,1989] **General Logics**, Proc. Logic Colloquium '87, edited by H.D. Ebbinghaus et al., North Holland, 1989.

[Scott, 1982] **Domains for Denotational Semantics**, in the proceedings of ICALP, Springer Lecture Notes in Computer Science 140 (1982).

CC^+ : an extension of the Calculus of Constructions with fixpoints

Philippe Audebaud

LaBRI - Université Bordeaux I - FRANCE
email : audebaud@labri.greco-prog.fr, paudebau@lip.ens-lyon.fr

August '92

Abstract

We follow an original idea suggested by Constable and Smith [CoSm87, CoSm88] providing a way for reasoning about non terminating computations in a typed framework. This work has been initiated within NuPrl by Smith [Smi88]. However its adaptation to the Calculus of Constructions (CC) has showed to be easy. We get a conservative extension of CC, denoted CC^+, where strong normalisation for β-reductions is preserved. We recover the alternate "recursive" coding for integers (already) introduced in AF2 by Parigot [Par88, Par92] ; thus the computational behaviour for the codes of integers is improved. Moreover, as expected, all partial recursive functions are definable. The proposition asserting that every term satisfies the principle of induction remains with no proof but trivial ones. All these results easily generalize to all the usual data structures.

1 Motivations

The initial Curry-Howard isomorphism has been greatly extended in the past decade, leading to powerful type systems. The correspondance between logic and functional programming within a typed framework forces both proofs and programs to be identified with strong normalizing lambda-terms.

Even if we may content ourselves with such a strong result from a pure logical point of view, its seems clear enough that type systems miss the point as far as they are intended to be considered as development systems for correct programs. For instance, treatment of exceptions, unbounded loops or recursive definitions are widely used in all programming languages in current use.

So, connections between logic and programming need to be refined, maybe even beyond Curry-Howard correspondance. Griffin's original paper [Gri90] emphasizes the interactions between continuations (in Scheme) and absurd reasoning. This is the starting point of studies undertaken, among other people, by Murthy [Mur90] and Krivine [Kri90a], leading to a somewhat different approach of the relation between the two domains.

From an other point of view, unbounded loops and recursives definitions seem to require the consideration of lambda-terms no longer normalisable. Starting from the the thesis that recursive structures can be coded by fixpoints, this paper is concerned with the study of an extension of a typed framework with a fixpoint constructor allowing us to examine more deeply how it should be possible to develop type systems where to reason with non terminating programs or other recusive program scheme. Our study, restrited to CC, actually develops in the same way as Parigot's study of recusive integers in the $\mathsf{AF2}$ framework, but independantly since originated by Constable and Smith papers and Paulin-Mohring's thesis [PaMo89c]. Moreover, most of the work remains to be done in order to answer the question whether or not such a solution catch most of the spirit beyond "recusive" features in current use in functional programming languages.

This paper develops as follow : in a first part we give a short overview of different possible solutions for the introduction of partial terms and we describe how to make the last of these solution fit with the Calculus of Constructions. Then, the extension CC^+ is presented and examined with its main metamathematical properties. The second part is devoted to the study of different internal codings for intergers in our framework ; there, computational and logical expressivity are the case in point.

2 Partial objects in a typed framework

From now on, we rely on the thesis that "recursive" structures can be coded through fixpoint constructs. Let $\mathcal{T}$ be any type system ; we assume at least the $\rightarrow$ constructor is available in $\mathcal{T}$. The main property we should expect from a type system is its logical consistency : there exists at least one type inhabited by no term.

2.1 A first attempt

Let P be any type and 0 an empty type. A fixpoint constructor, say $fix()$, given a term $f : P \rightarrow P$, provides a new term $fix(f) : P$. One step of computation obviously consists in an unfolding step

[βfix] $$fix(f) \longrightarrow f(fix(f))$$

Now, observe that, we can always form $\lambda x.x : P \rightarrow P$; hence $fix(\lambda x.x) : P$. In particuliar, the type 0 can no longer remains a empty type. Consistency is lost.

2.2 A refinement

We may wonder whether it is possible to avoid this phenomenon by allowing such a new term to be formed only in case P is already an inhabited type. Although there is no general answer to such a question, this refinement fails, for example in the Intuitionnistic Type Theories (ITT). Per Martin-Löf observed that the base type N is now provided with a fixpoint for the successor function. But then the axiom $\forall n \; n \neq n+1$ is no longer valid.

2.3 A general solution

In [CoSm87, CoSm88], Constable and Smith suggested a simple and elegant solution which avoids the difficulties encountered. Moreover, it seems to fit with most of the type systems.

Starting from a type system $\mathcal{T}$, we build an extension $\mathcal{T}'$ where the same rules are allowed at the level of types and terms as well. However a new feature is now available. Each type P is associated a new **bar** type : $\overline{P}$. This type is intended to be the type of the computations over terms belonging to P ; these computations, when converging (having a value) denote terms from P. Thus the fundamental rule, from the authors' point of view, is :

[bar] $$a \in \overline{A} \iff (a\downarrow \Rightarrow a \in A)$$

where $t\downarrow$ is an internal predicate asserting the convergence of t.

Let us give some examples.

$A \to \overline{A}$ the type for partial functions over A.

$\overline{A \to A}$ the type for terms which, if they converge, are total functions.

$\overline{\overline{A} \to \overline{A}}$ which corresponds to the type of all partial functions over A in a lazy programming language.

These examples suggest that we can expect to work in a very expressive type theory, from the point of view of the degrees of partiality that can be expressed.

As expected, the introduction of fixpoints is now allowed over bar types only :

[fix] $$\text{If } f \in \overline{A} \to \overline{A} \quad \text{then} \quad fix(f) \in \overline{A}$$

Some examples are provided in [CoSm87, CoSm88] and in [Smi88] where a consequent study of this kind of extended type system is done in the NuPrl framework. We devote our attention to the adaptation of this idea to a somewhat different type system.

2.4 Adaptation to the Calculus of Constructions

Although the adaptation from NuPrl to CC followed a more sinuous way, it is possible to explain the translation shortly. We are about to work with terms for which termination can be guaranteed no longer. This is what we want at the level of programs. Nevertheless, up to now, logical proofs and programs are all identified with the same objects, the same typed terms in the system. But we do want to consider that a logical proof must be "complete" so as to say. So, from the logical point of view, strong normalisation is the property required.

It seems clear for us that logical proofs and programs can no longer be identified. The notion of "value" is different from the two points of view. It is actually the case that the introduction of bar types proceeds from this analysis, even if the justification has not be given this way by Constable and Smith. So we have to find the right way to split the system into two parts : the first being semantically taken as the logical system, and the second one as the programming language. Now we already know how to do the job. In [PaMo89c], Paulin-Mohring introduced a new constant *Spec* (*Set* elsewhere) in order to distinguish between terms with or without any informative contents. The explanation, given explicitly, was to separate proofs from programs. And such a trick does the job, since then, terms are unambigous correspondance with one of the two constants *Prop* and *Spec* (or *Set*). Hence, the solution follows the same idea : we introduce a new constant, named $\overline{Prop}$, in order to mark terms to be considered as partial terms. Here is for the construction of bar types in the Calculus of Constructions. Now, partial programs and partial schema are easily formed through a single new constructor, in the CC-like style.

At least, we can accept a logical proof as soon as there is a proof for its termination. This last point is taken into account through the predicate of convergence within NuPrl. However we consider that this predicate cannot be put in the system ; but rather belongs to a metalevel with respect to CC system. So this part will be dropped out from our extension.

3 Presentation of CC$^+$; main results

Although we need only the new constant $\overline{Prop}$, the use of the additional symbol $\overline{Type}$ will ease presentation. However, $\overline{Type}$ and $Type$ can be confuse in any implementation of our extension.

3.1 Terms

The set of terms of CC^+, denoted Λ^+, is the least set of terms containing the constants $Prop$ and $\overline{Prop}$, a denumerable set $\mathcal{V}$ of variables, and close for the following constructs :

$$\begin{array}{rl} \text{application} & (M\ N) \\ \text{abstraction} & [x:M]N \\ \text{fixpoint} & \langle x:M\rangle N \\ \text{product} & (x:M)N \end{array}$$

where $M,\ N \in \Lambda^+$ and $x \in \mathcal{V}$.

3.2 Positive occurrences of a variable

The necessity of a restriction on the recusive schema is obvious, if we expect to keep the calculus as well behaved as possible. Otherwise, for instance, the proposition

$$\Lambda \equiv \langle X : \overline{Prop}\rangle(C : \overline{Prop})((X \to X) \to C) \to C$$

will collect all pure lambda-terms !

Let $M \in \Lambda^+$ and $x \in \mathcal{V}$ a fixed variable. The predicates $Pos_X(\cdot)$ and $Neg_X(\cdot)$ are given a mutually recursive definition as follow :

- if $X \notin FV(M)$ then $Pos_X(M)$ and $Neg_X(M)$;
- if $M \equiv X$ then $Pos_X(M)$;
- if $M \equiv (B\ A)$ or $[y:A]B$ then
  - $Pos_X(M)$ if $X \notin FV(A)$ and $Pos_X(B)$,
  - $Neg_X(M)$ if $X \notin FV(A)$ and $Neg_X(B)$.
- if $M \equiv (y:A)B$ then
  - $Pos_X(M)$ if $Neg_X(A)$ and $Pos_X(B)$,
  - $Neg_X(M)$ if $Pos_X(A)$ and $Neg_X(B)$;
- if $M \equiv \langle y:A\rangle B$ then
  - $Pos_X(M)$ and $Neg_X(M)$ if $X \notin FV(M)$.

The case for a fixpoint is justified plainly in the above lines.

3.3 Notions of reduction

Substitution is defined straightforwardly. There are two notions of reduction:

$$\begin{array}{ll} \beta & ([x:M]N \quad L) \longrightarrow_{\beta} N[x/L] \\ \beta fix & \langle x:M\rangle N \longrightarrow_{\beta fix} ([x:M]N \quad \langle x:M\rangle N) \end{array}$$

where $M, N, L \in \Lambda^{+}$ and $x \in \mathcal{V}$. We let $\rightarrow_{+}^{*}$ be the reflexive and transitive closure of $+ \equiv \beta \cup \beta$fix and $=_{+}$ the congruence generated by it.

Our choice for βfix comes from the fact that if ω is a fixpoint for the function f then we do have $\omega = f(\omega)$. Choosing βfix as $\langle x:M\rangle N \longrightarrow_{\beta fix} N[x/\langle x:M\rangle N]$ would have hidden the fact that, in the Calculus of Constructions, all the fixpoints we may construct are actually fixpoints of functional terms. Moreover it would have led to a bad reduction behavior. For, dealing with fixpoint computations involves that we must take *head* reductions as meaningful.

Notice that the condition imposed to a fixpoint, for the positivity of a variable in this fixpoint, can be deduced from the cases for the application and the abstraction. This a strong condition but there is no way to fix a weaker one since we do not know anything on the properties of the calculus. Then, in order to prove good properties on the calculus, we need to fix a condition invariant under reduction, hence under βfix reduction in the present case.

Then, both reduction is substitutive and the reduction + is weakly confluent (weakly Church-Rosser).

3.4 Inference and equality rules

Figures 1 and 2 recall the rules valid in CC. The letter $\mathcal{S}$ stands for any constant *Prop*, *Type* ... and $\overline{Prop}$, $\overline{Type}$ as far as CC^{+} is concerned. Notice the shortcut { } for any of the constructs [ ], ( ) or $\langle\ \rangle$.

The additional rules required in $\mathsf{CC}^{+}$are given in figure 3.

3.5 Syntax of a well typed term

Any well-formed term M is of one of the following forms (modulo α-conversion):

$$M \equiv [\overrightarrow{x:P}](\overrightarrow{y:Q})(M' \quad \overrightarrow{R})$$

with

$$M' \equiv \begin{cases} ([z:T]M_1 \quad M_2) \\ \langle z:T\rangle M_1 \\ \text{constant or variable} \end{cases}$$

Where $\overrightarrow{x:M}$ is for $x_1 : M_1 \ldots x_k : M_k$, $k \geq 0$.

Moreover, if M is a well typed term, of type N say, then we shall say that M is **total** if either $N \equiv Type$ or N is of type *Type* or *Prop*. Otherwise such a term is said to be **partial**. We denote CC^{+}_{tot} and CC^{+}_{par} the two sets of the partition of well typed terms in CC^{+}. It is easily checked that this definition does not depend on N, hence providing a new invariant for the (well typed) terms, as expected from the section 2.4.

empty $$\frac{}{[]\ \text{is valid}}$$

var-intro $$\frac{\Delta \vdash M \in \mathcal{K} \quad x \in \mathcal{V} \quad x \notin \Delta}{\Delta,\ x : M\ \text{is valid}}$$

hypothesis $$\frac{\Delta \vdash \quad x \in \mathcal{V}(\Delta)}{\Delta \vdash x \in \Delta(x)}$$

Prop-intro $$\frac{\Delta \vdash}{\Delta \vdash \mathit{Prop} \in \mathit{Type}}$$

prod-intro $$\frac{\Delta,\ x : P \vdash M \in \mathcal{S}}{\Delta \vdash (x : P)M \in \mathcal{S}}$$

abs-intro $$\frac{\Delta,\ x : P \vdash M \in N}{\Delta \vdash [x : P]M \in (x : P)N}$$

appl $$\frac{\Delta \vdash M \in (x : P)N \quad \Delta \vdash R \in Q \quad \Delta \vdash P = Q}{\Delta \vdash (M \quad R) \in N[x/R]}$$

type-conv $$\frac{\Delta \vdash M \in N \quad \Delta \vdash N = N'}{\Delta \vdash M \in N'}$$

Figure 1: Rules of inference for CC

refl
$$\frac{\Delta \vdash M \in P}{\Delta \vdash M = M}$$

sym
$$\frac{\Delta \vdash M = N}{\Delta \vdash N = M}$$

trans
$$\frac{\Delta \vdash M = N \quad \Delta \vdash N = P}{\Delta \vdash M = P}$$

cons
$$\frac{\Delta \vdash P = Q \quad \Delta,\ x : P \vdash M = N}{\Delta \vdash \{x : P\}M = \{x : Q\}N}$$

eqapp
$$\frac{\Delta \vdash (M\ N) \in P \quad \Delta \vdash M = M' \quad \Delta \vdash N = N'}{\Delta \vdash (M\ N) = (M'\ N')}$$

$\boldsymbol{\beta}$
$$\frac{\Delta,\ x : P \vdash M \in N \quad \Delta \vdash R \in P' \quad \Delta \vdash P = P'}{\Delta \vdash ([x : P]M\ R) = M[x/R]}$$

Figure 2: Equality rules for CC

$\overline{\textbf{Prop}}$**-intro**
$$\frac{\Delta \vdash}{\Delta \vdash \overline{Prop} \in \overline{Type}}$$

fix-intro-$\overline{Prop}$
$$\frac{\Delta \vdash P \in \overline{Prop} \quad \Delta,\ x : P \vdash M \in Q \quad \Delta \vdash P = Q}{\Delta \vdash \langle x : P \rangle M \in P}$$

fix-intro-$\overline{Type}$
$$\frac{\Delta \vdash P \in \overline{Type} \quad \Delta,\ x : P \vdash M \in P \quad \text{such thatsuch that } Pos_x(M)}{\Delta \vdash \langle x : P \rangle M \in P}$$

$\boldsymbol{\beta}$**-fix**
$$\frac{\Delta,\ x : P \vdash M \in P}{\Delta \vdash \langle x : P \rangle M = ([x : P]M\ \langle x : P \rangle M)}$$

Figure 3: Additional rules of inference and equality for CC^+

3.6 Main results

conservativity CC^+ is a conservative extension of CC, hence is consistent. (Hint : there is an obvious extraction function from CC^+_{tot} to CC.)

uniform properties The following properties are true over the set of well typed terms :

- Confluence for the reduction $+ = \beta \cup \beta fix$;
- Strong normalisation for β reduction ;
- Finiteness of developments and standardisation theorems.

For hints see [Aud91] ; for a complete treatment see [Aud92, Aud93].

3.7 Terms "almost in normal form"

The set of normal forms (NF) is too much restrictive since, for example, the term $[x : P]x$ where $P \equiv \langle X : \overline{Prop} \rangle (C : \overline{Prop})(X \to C) \to C$, or any other fixpoint, would be excluded otherwise. However, the type information provided by the "abstraction" parts of a term has no computational meaning ; it is dropped down in the pure lambda-term obtained through the erasure function.

Then we can define a weaker notion of normal form, well suited with an environment of partial terms. The set ANF of terms **almost in normal form** is defined as the least class of well-formed terms containing the constants $Prop$ and $\overline{Prop}$, the set $\mathcal{V}$ of variables and closed by

$$M \text{ ANF} \quad \Rightarrow \quad [x : P]M \text{ ANF}$$

$$\begin{array}{rcl} M \text{ ANF} & \Rightarrow & (x:P)M \text{ ANF} \\ N_1 \cdots N_k \text{ ANF} & \Rightarrow & (x\ N_1\ \cdots\ N_k) \text{ ANF} \end{array}$$

Where $P \in \Lambda^+$ being any type and x is any constant or variable.

We get the obvious facts :

- any propositional type is an ANF ;
- NF $\subseteq$ ANF $\subseteq$ HNF.

The terms ANF will play the same rôle as NF terms in CC, say, as will now be seen through the different codings for integers in the extension CC^+.

4 Internal codings for integers

This part is devoted to the question how both computational and logical expressivity are possibly improved in CC^+. The main originality of this extension is the ability to give an alternate coding for integers. A complete treatment is already given in [Aud91, Aud92] for CC^+, and in [Par88, Par92] for $\mathsf{AF2}$. So we would rather like giving a somewhat informal presentation for to major codings : integers as iterators and as selectors. We emphasize our presentation does not claim to be rigorous. However, we expect this approach to show how the computational behavior of (codes of) integers uniquely depends upon a more or less immediate proof of mathematical equality.

4.1 Primitive recursion over integers

Given $a \in A$ and $h \in \mathsf{N} \times A \to A$, the primitive recursion schema says there exists a (unique) solution $g \in \mathsf{N} \to A$ such that :

$$g(0) = a \quad \text{and} \quad \forall n \in \mathsf{N}\ g(n+1) = h(n, g(n))$$

Informally :

$$\begin{tikzcd}
& \mathsf{N} \arrow[r, "s"] \arrow[dd, "\langle f,g\rangle"] & \mathsf{N} \arrow[dd, "\langle f,g\rangle"] \\
\mathbf{1} \arrow[ur, "0"] \arrow[dr, "a"'] & & \\
& \mathsf{N} \times A \arrow[r, "\langle \pi_1 \circ s, h\rangle"'] & \mathsf{N} \times A
\end{tikzcd}$$

As a consequence, $f : \mathsf{N} \to \mathsf{N}$ must satisfy :

$$f(0) = 0 \quad \text{and} \quad f \circ s = s \circ f$$

Thus, in any cartesian closed category with N as a natural number object (nno), we can conclude $f = id_{\mathsf{N}}$. It works since N is precisely an initial object $\mathbf{1} \xrightarrow{0} \mathsf{N} \xrightarrow{s} \mathsf{N}$ in the category of diagrams $\mathbf{1} \xrightarrow{x} C \xrightarrow{f} C$.

4.2 Integers as iterators

In CC (in system F already), integers are internally coded in such a way we collect Church's integers as normal forms through the proposition :

$$\mathsf{Nat}^i \equiv (C : Prop)C \rightarrow (C \rightarrow C) \rightarrow C \in Prop$$

Thus this representation mimics as well as possible the position of Nat^i as nno in the type system. Let 0^i and S^i be the closed terms which code zero and the successor function. Then the "regular integers" are coded through the set $\{((S^i)^k\ 0^i)k \in \mathsf{N}\}$. The induction principle is :

$$Ind^i \equiv [n : \mathsf{Nat}^i](P : \mathsf{Nat}^i \rightarrow Prop)(P\ 0^i) \rightarrow ((n : \mathsf{Nat}^i)(P\ n) \rightarrow (P\ (S^i\ n))) \rightarrow (P\ n)$$

This principle expresses that any property is true for an integer, as soon as it is true on the set of codes of regular integers.

Two major problems are carried out by such a representation :

- No proof exists for $(n : \mathsf{Nat}^i)(Ind^i\ n)$. As noticed in [PaMo89a, PaMo89c], we need the fact that $(n\ \mathsf{Nat}^i\ 0^i\ S^i)$ equals n. But this property means precisely that every $n : \mathsf{Nat}^i$ is actually the code for a regular integer. n being a variable, this is obviously impossible to know. However this equality holds for closed terms, at the meta level. Here is for the structural point of view, as far as logical expressivity is concerned.

- Now, let us have a look at the computational expressivity. A solution to primitive recursion schema is provided in CC, through the recursor term rec^i, by

  $$g \equiv [n : \mathsf{Nat}^i](rec^i\ n\ A\ a\ h)$$

  Nevertheless the equality $f = id_{\mathsf{Nat}^i}$ can only be proved for regular integers, that is to say terms satisfying the induction principle. We need the same property : "n is zero or the successor of another term", but at the level of programs now ; this is a non dependent version for the former problem (see [PaMo89c, section 4.4.1]).

  So let us considerer we are dealing with those well built integers. We want to consider the simplest of all primitive recursion schema

  $$p(0) = 0 \qquad \text{and} \qquad p(n+1) = n$$

  giving the predecessor function. From the computational point of view, we get the following diagram

  $$\begin{array}{ccc} n & \longrightarrow & n+1 \\ \downarrow & & \downarrow \\ \langle f(n), p(n)\rangle & \longrightarrow & \langle f(n+1), n\rangle \end{array}$$

  Hence computation of $p(n+1)$ forces that of $f \circ s(n)$, so the computation of $s \circ f(n)$ too. Eventually, the effective computation of that term will have force that of $s \circ f(n), \ldots, s^n \circ f(0)$, even though we already know the result will be n. Thus this computation takes a number of β-réductions in $\bigcirc(n)$. This simple example makes it apparent the evaluation mecanism linked to the very nature of Church's integers : there are iterators. And, at the level of programs, their imperfection lies in that character.

How far is it possible to improve these two points ? In [CoPa89] Coquand and Paulin-Mohring gave a solution through an extension of CC with inductive types. We restrict our attention to CC^+.

4.3 Integers as selectors

In a cartesian closed category, any nno N satisfies another diagram :

$$\begin{tikzcd} \mathbf{1} \arrow[r, "0"] \arrow[dr, "x"'] & \mathsf{N} \arrow[d, "{[x,f]}"] & \mathsf{N} \arrow[l, "s"'] \arrow[dl, "f"] \\ & C & \end{tikzcd}$$

Let us consider the predecessor function. We note that $C \equiv \mathsf{N}$, $f \equiv id_{\mathsf{N}}$ and $x \equiv 0$. Then we get :

$$[id, 0](0) = 0 \quad \text{and} \quad [id, 0](n+1) = n$$

Clearly, $[id, 0]$ is the predecessor function over N. Moreover its computation is now immediate.

Although Nat^i satisfies this kind of diagram too, at least as far as we restrict ourselves to the set of well built terms. But then we get rather the existence of two terms such that : $\mathsf{Nat}^i \underset{\mathbf{in}}{\overset{\mathbf{out}}{\rightleftharpoons}} (C : Prop)C \to (\mathsf{Nat}^i \to C) \to C$ with $\mathbf{in} \circ \mathbf{out} = id$ and $\mathbf{out} \circ \mathbf{in} = id$.

Now, through fixpoint construction, we are able to define Nat^r such that

$$\mathsf{Nat}^r = (C : Prop)C \to (\mathsf{Nat}^r \to C) \to C$$

It suffices to define

$$\mathsf{Nat}^r \equiv \langle N : \overline{Prop}\rangle(C : \overline{Prop})C \to (N \to C) \to C$$

Notice that we must work over $\overline{Prop}$. Then we get

$$\begin{array}{lcl} 0^r & \equiv & [C : \overline{Prop}][x : C][f : \mathsf{Nat}^r \to C]x \\ S^r & \equiv & [n : \mathsf{Nat}^r][C : \overline{Prop}][x : C][f : \mathsf{Nat}^r \to C](f\ n) \\ P^r & \equiv & [n : \mathsf{Nat}^r](n\ \mathsf{Nat}^r\ 0\ id_{\mathsf{Nat}^r}) \end{array}$$

where P^r is the predecessor function.

More generally, a solution to any primitive recursion schema is easily found using fixpoint at the level of programs :

$$g = [n : \mathsf{Nat}^r](n\ A\ a\ [p : \mathsf{Nat}^r](h\ p\ (g\ p)))$$

This point clearly shows that introduction of fixpoints is a good way to improve the computational behaviour of integers. Moreover fixpoints provide more programs, that is more realizations for specifications of programs ; indeed we are not forced to give a solution g as a fixpoint. Nevertheless they are required in order to give a solution for μ-recursion schema : given $f : \mathsf{N} \to \mathsf{N}$, find the least integer n such that $f(n) = 0$. If we did not know there is such an integer, no solution exists in CC, since any computation terminates. In CC^+ the solution is obvious : take $g(0)$ where

$$g(k) = (f(k)\ \mathsf{Nat}\ k\ [p : \mathsf{Nat}]g(p+1))$$

in a informal syntax.

Main results

Let us the give here the main results about the "recursive" solution Nat^r. In the following lines, $\underline{n}$ is for the code of the nth integer.

- $\forall n \in \mathsf{N}\ (P^r\ (S^r\ \underline{n})) \rightarrow^* \underline{n}$ (in a fixed number of step indeed) ;
- The set of closed anf terms of type Nat^r is the set $\{\underline{n} \mid n \in \mathsf{N}\}$;
- All partial recursive functions over integers are definable in CC^+. (Hint : the proof follows the same lines as Barendregt's in **??**)

Logical expressivity

There remains the question whether or not CC^+ provides an improvement at the structural level. We pointed out Nat^i behaves as well as a nno provided it is restricted to the subset of well built terms. Since then Nat^i satisfies the second diagram, there must exist a correspondance between Nat^i and Nat^r. And it is actually the case through the terms :

$$\begin{array}{rcl} i & \equiv & [n : \mathsf{Nat}^i](n\ \mathsf{Nat}^r\ 0^r\ S^r) \\ & \in & \mathsf{Nat}^i \rightarrow \mathsf{Nat}^r \\ r & \equiv & [n : \mathsf{Nat}^r](n\ \mathsf{Nat}^i\ 0^i\ [p : \mathsf{Nat}^r](S^i\ (r\ p))) \\ & \in & \mathsf{Nat}^r \rightarrow \mathsf{Nat}^i \end{array}$$

Precisely, these two terms give a one-to-one correspondance between codes of regular integers in both of the types. But the fact Nat^i satisfies the same diagram as Nat^r says more. If it were to exist a proof for $(n : \mathsf{Nat}^r)(Ind^r\ n)$ then, inevitably, $(n : \mathsf{Nat}^i)(Ind^i\ n)$ would be provable too. The converse is true of course. And indeed we can prove :

(i) $(n : \mathsf{Nat}^i)(Ind^i\ n) \rightarrow (Ind^r\ (i\ n))$

(ii) $(n : \mathsf{Nat}^r)(Ind^r\ n) \rightarrow (Ind^i\ (r\ n))$

(iii) $(n : \mathsf{Nat}^i)(Ind^i\ n) \rightarrow n = (r \circ i\ n)$

(iv) $(n : \mathsf{Nat}^r)(Ind^r\ n) \rightarrow n = (i \circ r\ n)$

Of course, the very reason why it works is easy to understand. The induction principle, when satisfied, fills the gap between the subset of regular integers and all integers.

We are allowed to conclude the study of integers we took as an example, shows that our extension cannot be expected to improve the logical expressivity of the Calculus of Constructions.

5 Further developments

The study briefly undertaken in the case of the integers through various internal codings can be followed in its main lines as far as we restrict ourselves to the usual data structures such that lists or trees. Actually, close connections have been established between "simple (concrete) types" as described in [PaMo89c] and their recusive counterpart in CC^+ : see [Aud92]. Such an equivalence emphasizes the main result that introduction of fixpoints, the way it is done in

CC^+ at least, does not improve the logical power of CC but rather gives the user the ability of realizing more programs and more specificationss.

The better that kind of "limitation" is understood, the more we are able to simplify, to relax the rules previously given for the introduction of fixpoints. For example, Ch. Paulin-Mohring suggests to allow fixpoints within $Prop$ (hence dropping out $\overline{Prop}$) and, at the same time, to remove fixpoints at the level of proofs. This idea, and other variations, clearly deserve further attention.

As a matter of temporary conclusion let us focus the attention on a major theme, from our own point of view. Where mathematical reasoning usually contents itself with equalities, programming practice, thus usage of proof development systems as well, is mostly concerned with the way mathematical objects are computed. This dynamic point of view seems to belong to the core of this domain of research, if to be considered really as a science. Hence, we think that this difficulty should not be avoided. To give just one concrete example, fixpoints are allowed in CC^+ exactly where syntactic transformations already exist in CC. We got the feeling, a posteriori, that no fixpoint over schema would have been even conceivable otherwise. And, in that case, working with recursive schema means forcing the corresponding transformation to be taken as the identity function (cœrcion) ; at least this is true as far as we restrict ourselves to the set of "well built" terms within a simple type : see [PaMo89c].

References

[Aud91] Audebaud P. (1991) Partial Objects in the Calculus of Constructions. In *6nd Conf. on Logic in Comp. Science*. IEEE, 1991. Available through anonymous ftp to geocub.greco-prog.fr in /pub/Papers/Audebaud repertory.

[Aud92] Audebaud P. (1992) Extension du Calcul des Constructions par points fixes. Thèse, Université Bordeaux I, 1992. Available through anonymous ftp to geocub.greco-prog.fr in /pub/Papers/Audebaud repertory.

[Aud93] Audebaud P. (1993) Uniform properties in the Calculus of Constructions. *submitted* to TLCA'93.

[Bar84] Barendregt H.P. (1984) *The Lambda-calculus: its syntax and semantics.* 2nd ed. Norh-Holland. Amsterdam.

[CoPa89] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88.* LNCS **417**, Springer Verlag 1990.

[CoSm87] Constable R.L. and Smith S.F. (1987) Partial objects in constructive type theory. In *2nd Conf. on Logic in Comp. Science.* IEEE, 1987.

[CoSm88] Constable R.L. and Smith S.F. (1988) Computational foundations of basic recursive function theory. In *3nd Conf. on Logic in Comp. Science.* IEEE, 1988.

[CoMe85] Constable R.L. and Mendler N.P. (1985) Recursive definitions in Type Theory. In *Logics of Programs.* Parikh R. editor. LNCS **193**, Springer Verlag, 1985.

[Gri90] Griffin T. 1990. A formulæ-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Synposium on Principles of Programming Languages*, 1990.

[Kri90a] Krivine J.-L. 1990. Opérateurs de Mise en Mémoire et Traduction de Gödel. Rapport technique de l'Equipe de Logique. Université Paris VII. Janv. 1990.

[Kri90b] Krivine J.-L. 1990. Lambda-calcul, évaluation paresseuse et mise en mémoire. A paraître dans Informatique Théorique et Applications, RAIRO.

[Luo90] Luo Z. (1990) ECC, an Extended Calculus of Constructions. In *Information and Computation.*

[Mend87] Mendler N. (1987) Recursive Types and type Constraints in Second-Order Lambda Calculus. In *2nd Conf. on Logic in Comp. Science.* IEEE, 1987.

[Mur90] Murthy C. 1990. Extracting Constructive Content from Classical Proofs. Ph.D. Cornell University, 1990.

[Par88] Parigot M. (1988) Programming with proofs: a second order type theory. *Proceedings of ESOP '88 (ed. H. Ganziger)* Lectures Notes in Comp. Science **300**. Springer Verlag. Berlin.

[Par92] Parigot M. 1992. Recursive Programming with Proofs. Theoretical Computer Science **94** (1992) p.335-356.

[PaMo89a] Paulin-Mohring Ch. (1989) Extracting $F\omega$ programs from proofs in the Calculus of Constructions. In *Proceedings of 16th ACM Symposium on Principles of Programming Languages.* ACM, New-York.

[PaMo89b] Paulin-Mohring Ch. (1989) Inductive definitions in the Calculus of Constructions. (draft) In *The Calculus of Constructions* Rapport technique INRIA **110**.

[PaMo89c] Paulin-Mohring Ch. (1989) Extraction de programmes dans le Calcul des Constructions. Thèse Paris 7, 1989.

[Smi88] Smith S.F. (1988) Partial Objects in Type Theory. Ph.D. Dissertation. Cornell University.

Continuations and Simple Types: a Strong Normalization Result

Franco Barbanera
Torino

Abstract

We present a work in which we study the termination problem for a typed lambda-calculus with continuations. We do not bound ourselves to study a particular reduction strategy, like call-by-value or call-by-name. Reductions may be applied to any part of any term in any order. Our main result is that every reduction sequence in the system terminates.

Game theory and Program extraction.

Stefano Berardi
Torino

Abstract

We interpret each classical formula P as a game G(P) between two players (A and B). Player A is committed to show that P is true and player B that P is false. We interpret a proof as a recursive winning strategy for G(P). This work is inspired by Coquand's last work, differing from it in three aspects :

- we interpret linear logic first
- we use perfect information games
- we prove cut elimination in a purely intuitionistic way.

Deliverables: a categorical approach to program development in type theory

Rod Burstall and James McKinna*
Laboratory for the Foundations of Computer Science
University of Edinburgh†

August 11, 1992

1 Introduction

This paper outlines a method for constructing a program together with a proof of its correctness with respect to a given specification. The technology used is type theory, in fact an extended version of Calculus of Constructions Luo's ECC [22] as implemented in Pollack's 'Lego' system [24]; here *program* means a primitive recursive description of a function using simply typed lambda calculus with higher order functions and data types such as numbers and lists.

Given a precondition and a postcondition we consider pairs consisting of

- a program
- a proof that the program satisfies the postcondition given the precondition.

We call such a pair together with the pre- and postconditions a *deliverable*, since it is what a software house should ideally deliver to its client instead of just a program.

Now we observe that various operations, composition, pairing, iteration and so on, can be used to combine deliverables, and that these operations can be implemented in ECC. Consider for example composition. If (f, p) is a deliverable from pre to $post$ and $(f\prime, p\prime)$ is a deliverable from $post$ to $post\prime$ then their composition is a deliverable from pre to $post\prime$ thus

- $f \circ f\prime$, the composition of the functions
- a proof using p and $p\prime$ that $f \circ f\prime$ satisfies $post\prime$ given pre.

In fact the combining operations (excluding iteration) are exactly those of a cartesian closed category whose objects are the pre- and postconditions and whose arrows are deliverables. This is comforting since it assures us that we have a complete set of combinators. They enable us to build a complicated program together with its correctness proof. Since the resulting expression of the calculus denotes a pair we can extract the program trivially by evaluating the first element of the pair.

*The authors are gratefully for the support of the EC Logical Frameworks BRA and the SERC
†J.C.M.B., King's Buildings, Mayfield Rd., Edinburgh EH9 3JZ, UK;
e-mail `rb@dcs.ed.ac.uk, jhm@dcs.ed.ac.uk`

We have used the Lego system to define these combinators and use them to create the deliverables for several programs as examples.

However in many cases a precondition and a postcondition do not satisfactorily specify a desired program. For example when developing a sorting program the precondition might ask for an arbitrary list and the postcondition demand that the result be a sorted list. But we would get little thanks for a program which always returned the nil list, even though this is sorted. We need to specify that the output list should also be a permutation of the input one.

If we try to replace the pre- and postconditions by a single statement specifying a relation between input and output there do not seem to be convenient combination operators; so we discard that approach. Thinking syntactically, we could allow the pre- and postconditions to share a free variable. For example using a free variable l for lists

$$pre(x) = x = l$$

$$post(y) = isPermutation(y, l)$$

This ties the pre- and postconditions together, but such free variables should have a definite scope.

We can think, more semantically, that l satisfies some base condition (it must be a list) and the pre- and post conditions are relations whose first argument is l. Thus we can talk of pre- and post conditions *relative to* l. Now a deliverable relative to l is a program which takes the precondition to the postcondition for all l. We will refer to these as *second order* specifications and deliverables in contrast to the original *first order* ones.

How should we represent specifications and deliverables in type theory? We choose the following.

A first order specification is a predicate over a type thus

- a type, s
- a function, S, from s to $Prop$ (the type of propositions)

and a first order deliverable from specification (s, S) to specification (t, T) is a program with proof thus

- a function , $f : s \to t$
- a proof, $\phi : \forall x : s.Sx \to T(fx)$ where $\forall$ is the $\prod$ quantifier on propositions.

Now we can generalise these to work relative to a first order specification (u, U), which corresponds to the free variable l in our example. Call u the base type; now the second order specification becomes a relation over the base type and another type, and the second order deliverable must respect the pre- and postrelations. We leave the details to the body of the paper.

We might try to characterise the first order deliverables as a cartesian closed category whose objects are specifications and whose arrows are deliverables; this is almost correct, but we have to modify the CCC to a *semi*-cartesian closed category, a mild generalisation due to Hayashi [12]. The second order deliverables are then characterised using a fibration whose fibres are categories of first order deliverables. In fact this structure constitutes a model of type theory.

The paper includes some illustrative examples of program development using deliverables and brief mention of others, of which the most substantial is an algorithm for the Chinese Remainder theorem.

We also discuss the relation to earlier work on extracting programs from proofs, particularly the work of Martin-Löf as developed by Nördström, Petersson and Smith [29]. They define a translation from a type theory, say S, to an underlying type theory U such that the types in S correspond to our specifications and the unary terms in S correspond to our deliverables. A somewhat similar approach is taken by Paulin-Mohring [30] in her program extraction using the realisability interpretation.

The virtue of our method is that it can quite easily be coded up in an existing system, and there is no difficulty about extracting the program from the proof, since they are built together but as separate components of a pair; normalising the first component gives us the program. We have mentioned operations for combining deliverables, and this might suggest a bottom-up approach; but we can just as well start with a specification of the desired program and derive the corresponding deliverable top-down by refinement in Lego.

The weakness of our method is that we have to use a combinatory notation which is less perspicuous than the usual λ notation of type theory. Since deliverables form a model of type theory, this might possibly be rectified by extending ECC with an extra kind whose types would be specifications and whose terms would be deliverables, thus following the insight of Martin-Löf; but we have not yet worked out this approach.

The basic approach was proposed in an unpublished talk by Burstall at the Båstad Workshop on type theory in 1988. A brief account by the present authors appeared as [4], and a full treatment of work so far is in McKinna's thesis [25].

2 Review of ECC and a simple example

Luo's Extended Calculus of Constructions, ECC, [21, 22] is a rich type theory containing Coquand and Huet's Calculus of Constructions [6, 7] as a subsystem, together with strong Σ-types and a cumulative hierarchy of predicative universes, much as in the systems considered by Martin-Löf and his collaborators [26, 27, 29]. In Martin-Löf systems, *all* types may be read as propositions, and in Coquand and Huet's original system, all propositions may be read as types. ECC avoids this blurring of distinctions, giving us access to full intuitionistic higher-order logic at a propositional level, together with a predicative environment for computation and abstract mathematics. It is precisely this ability to distinguish propositional from computational information within a single framework which underlies our approach to program development.

ECC is built out of a calculus of terms with the following official syntax

$$\begin{array}{rcl} T & ::= & \kappa \mid V \mid \\ & & \Pi V{:}T.T \mid \lambda V{:}T.T \mid TT \mid \\ & & \Sigma V{:}T.T \mid \mathbf{pair}_T(T,T) \mid \pi_1(T) \mid \pi_2(T) \end{array}$$

where V ranges over some infinite collection of variables, and κ ranges over Prop and Type_i ($i \in \omega$), the so-called *kinds* of ECC. Prop is an impredicative universe, as in Coquand and Huet's original systems [6], intended to contain propositions, while the Type_i are predicative universes much like a set-theoretic hierarchy (and very similar to the $\mathbf{U}_i$ of some versions of Martin-Löf's theories [29, 27]). Substitution for free occurrences of variables is defined in the usual way. Terms are identified up to renaming of bound variables. The basic conversion relation $\simeq_\beta$ is defined on all terms, and as usual is the congruence closure of the familiar reductions:

(β) $(\lambda x{:}A.M)N \rhd M[N/x]$, and

{V:T}T	corresponds to	$\Pi x \colon T.\, T$,
[V:T]T	to	$\lambda x{:}T.\, T$,
<V:T>T	to	$\Sigma V{:}T.T$,
(T,T)	to	$\mathbf{pair}_{\Sigma x:A.B}(T,T)$,
T.1	to	$\pi_1(T)$,
and T.2	to	$\pi_2(T)$.

Table 1: Comparison between syntax of LEGO and ECC

$(\sigma)\ \ \pi_i(\mathbf{pair}_T(M_1, M_2)) \rhd M_i (i = 1, 2)$.

LEGO is Pollack's implementation of a typechecker and refinement proof system for ECC and a number of related systems, based on earlier ideas of Huet, de Bruijn and others [3, 6].

The LEGO syntax for terms in the language of ECC is given by

```
T ::= Prop | Type(i) (i ∈ ω) | V
      {V:T}T | [V:T]T | TT |
      <V:T>T | (T,T) | T.1 | T.2
```

where the correspondence with the official syntax is given in Table 1. As usual, the non-dependent Π-types is denoted by an infix $\to$.

The syntax has recently been extended to provide arbitrary extensions to the conversion relation. As an example, we may add a type of natural numbers, by declaring appropriate constants in an initial context

$$\begin{aligned}\Gamma_{nat} \quad = \quad & nat{:}\,Type_0, 0{:}nat, S{:}nat \to nat,\\ & natrecd^1{:}\Pi C{:}nat \to Type.\ (C0) \to (\Pi k{:}nat.\ (Ck) \to (C(Sk))) \to \Pi n{:}nat.\ Cn\end{aligned}$$

together with reductions

$$\begin{aligned} natrecd\ C\ z\ s\ 0 \quad &\rhd \quad z \\ natrecd\ C\ z\ s\ (Sk) \quad &\rhd \quad s\ k\ (natrecd\ C\ z\ s\ k)\end{aligned}$$

for C, z, s, k of the appropriate types.

In this context, we may derive the following term

$$\Gamma_{nat}, n{:}nat \vdash \forall\Phi{:}nat \to Prop.\Phi(0) \Rightarrow (\forall k{:}nat.\Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(n) : Prop$$

which represents impredicatively the (informal) proposition that n is an *even* natural number. That is, we define even numbers to be those n which satisfy *all* predicates satisfied by 0 and closed under successor of successor (the impredicativity, of course, lies in the fact that "evenness" is just such a predicate). Moreover, in this representation

$$\begin{aligned}\Gamma_{nat} \quad \vdash \quad & \lambda\Phi{:}nat \to Prop.\\ & \quad \lambda z{:}\Phi(0)\,.\\ & \quad\quad \lambda s{:}\forall k{:}nat\,.\,\Phi(k) \Rightarrow \Phi(S(Sk))\,.\\ & \quad\quad\quad z\\ : \quad & \forall\Phi{:}nat \to Prop.\Phi(0) \Rightarrow (\forall k{:}nat.\Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(0)\end{aligned}$$

[1] "*d*" for dependent.

is a term representing a proof that 0 is even.

Example In the same way, one could define a predicate representing "oddness",

$$Odd =_{\text{def}} \lambda n{:}nat.\ \forall\Phi{:}nat \to Prop.\Phi(S0) \Rightarrow (\forall k{:}nat.\Phi(k) \Rightarrow \Phi(S(Sk))) \Rightarrow \Phi(n)$$

being one such representation. One might then show that the successor function S transforms even numbers into odds, and odds to evens, yielding terms sEO of type $\forall n{:}nat.\ Even(n) \to Odd(Sn)$ and sOE of type $\forall n{:}nat.\ Odd(n) \to Even(Sn)$ (the patient reader may care to experiment with the system by doing just such a thing).

What is the behaviour of the composite function $x \mapsto x+2$? We may use the proofs sEO and sOE, together with *modus ponens*, to derive a term $ssEE$ of type $\forall n{:}nat.\ Even(n) \to Even(n+2)$.

Now the Σ-types of the calculus allow us to pair together these fragments of code with their associated proof terms, for example

$$(S, sEO) : \Sigma f{:}nat \to nat.\ \forall n{:}nat.\ Even(n) \to Odd(fn)$$

and we may see the *pair* $(\lambda n{:}nat.\ n+2,\ ssEE)$ arising as a composite construction from the *pairs* (S, sEO) and (S, sOE).

The pair $(\lambda n{:}nat.\ n+2,\ ssEE)$ of type

$$\Sigma f{:}nat \to nat.\ \forall n{:}nat.\ Even(n) \to Even(fn)$$

is a very simple instance of the general idea below of a *first-order deliverable*, in this case from $(nat, Even)$ to $(nat, Even)$.

3 Definition of first-order deliverables

We work relative to some well-formed context Γ in ECC.

Definition 3.1 *specification*
A *specification* is a pair of terms s, S such that $\Gamma \vdash s : Type$, and $\Gamma, x{:}s \vdash Sx : Prop$.

We typically write $\mathcal{S}, \mathcal{T}, \mathcal{U}$ for specifications, and understand s, t, u (respectively S, T, U) as referring to their underlying type (resp. predicate). Formally, there is a type of specifications, namely $SPEC_1 =_{\text{def}} \Sigma s{:}Type.s \to Prop$, so that we may consider operations which construct specifications entirely within the framework of ECC (*cf.* the account of specifications and refinements in [23]).

We will consider a category whose objects are the specifications. Specifications defined by *logically* equivalent predicates in general define distinct objects. The appropriate notion of morphism between specifications consists of a pair (f, ϕ), where f is a function between the underlying types, and ϕ is a proof that f respects the predicates. We call these morphisms *first-order deliverables.*

Definition 3.2 *first-order deliverable*
Given specifications $\mathcal{S} = (s, S), \mathcal{T} = (t, T)$, a *first-order deliverable* is a term $\mathcal{F}$ such that

$$\Gamma \vdash \mathcal{F} : \Sigma f : s \longrightarrow t.\forall x : s.Sx \Longrightarrow T(fx).$$

The motivation for such a definition goes right back to Hoare's original paper on axiomatic semantics [14], and, like the logic of triples which bears his name, expresses in a *formal* system the informal notion of a program, together with a certificate of some specified input/output behaviour. Of course, we are concerned with a functional language, rather than an imperative one, so there is no confusion over program variables and logical variables. In this framework, moreover, the proof and the program are linked as a pair. This definition uses the Σ-types in an essential way to capture this idea. We may use all the features of ECC to construct such pairs, but based on our intuitions about computational *vs.* propositional information, we insist upon a trivial extraction process: first projection π_1 from the Σ-type yields the underlying algorithm f. Indeed this accounts for the name "deliverables": they are what a software house should deliver to its customers, a program plus a proof in a box with the specification printed on the lid (the Σ-type). The customer can independently check the proof and then run the program, without the need for a complicated extraction process which may yield an unusual algorithm. Indeed, we propose this method as a style for developing programs in the first place. We usually have a reasonable idea of the algorithm in advance, as opposed to its proof of termination, or even correctness, and we would like an understanding, which reflects our intuitions, of how to build up these deliverables from smaller ones, for example by refinement and composition, possibly with machine assistance.

In terms of LEGO, we define a predicate Del_1 and a type constructor $\mathbf{del}_1$ which describe first-order deliverables within ECC (compare Definition 3.2 above):

```
Lego> Del1;
value = [s,t|Type][S:Pred s][T:Pred t][f:s->t]{x|s}(S x)->T (f x)
type  = {s,t|Type}(Pred s)->(Pred t)->(s->t)->Prop
Lego> del1;
value = [s,t|Type][S:Pred s][T:Pred t]<f:s->t>Del1 S T f
type  = {s,t|Type}(Pred s)->(Pred t)->Type
```

(This is the output from the typechecker after giving the definitions). We have exploited the implicit syntax to enable us to suppress the argument types[2] in the predicate `Del1` and the type `del1`.

With an eye to the categorical aspects of this definition, we typically write

$$\mathcal{S} \xrightarrow{\mathcal{F}} \mathcal{T} \in \mathbf{del}_1 \quad \text{or} \quad (s,S) \xrightarrow{(f,\phi)} (t,T) \in \mathbf{del}_1$$

when $\Gamma \vdash f : s \longrightarrow t$, and $\Gamma \vdash \phi : \mathrm{Del}_1\ S\ T\ f$. Since s,t may be inferred by the typechecker, and we are in general not interested in ϕ, save to know that it exists, we may even abuse our notation and write

$$S \xrightarrow{f} T \in \mathbf{del}_1.$$

At this stage, we need not have used the Σ-types to present these definitions. However, internalising the mathematical pair in a Σ-type allows us to represent operations which produce such function-proof pairs within the calculus. This gives us the possibility of developing a structure on these morphisms.

[2]This accounts for our choice of notation: since the types `s,t` may be inferred from `S,T`, we relegate them notationally to the lower case.

Definition 3.3 *equality of specifications*
Given specifications $(s, S), (t, T)$, we say $(s, S) = (t, T)$ if

$$s \simeq_{\beta\delta} t \text{ and } \lambda x{:}s.\ S\,x \simeq_{\beta\delta} \lambda x{:}t.\ T\,x.$$

Definition 3.4 *equality of deliverables*
Given

$$(s, S) \overset{(f,\phi)}{\underset{(g,\psi)}{\rightrightarrows}} (t, T) \in \mathbf{del}_1,$$

we say $(f, \phi) = (g, \psi)$ if

$$\lambda x{:}s.\ fx \simeq_{\beta\delta} \lambda x{:}s.\ gx$$

and

$$\lambda x{:}s.\ \lambda h{:}S\,x.\ \phi x h \simeq_{\beta\delta} \lambda x{:}s.\ \lambda h{:}S\,x.\ \psi x h.$$

This definition of equality of deliverables seems to be the minimal extension of the basic conversion relation which ensures good categorical properties.

3.1 Semi-structure in categories

The use of cartesian closed categories to give models of the simply-typed λ-calculus is by now familiar in computer science [8, 20, for example], as are various equational presentations of the structure of a cartesian closed category. The basic type and term constructors are defined by adjunctions. In this analysis, the unit and counit of the adjunction defining the arrow type correspond, loosely, to η and β conversion, respectively (and similarly for the product type constructor). The absence of η-conversion and surjective pairing in ECC forces some extra technical difficulty upon us. However, models of various typed λ-calculi without η-conversion and surjective pairing can be given a rigorous semantic account in terms of *semi-adjunctions*, introduced by Hayashi in [12]. Essentially, the equations defining an adjunction are relaxed sufficiently that, under suitable conditions, there is a notion of counit which corresponds appropriately to β-conversion, without a unit corresponding to η-conversion. A detailed treatment is given in the thesis [25].

Definition 3.5 *Semi-functor*
Given two categories $\mathbf{C}, \mathbf{D}$, a *semi-functor* between them is "a functor which need not preserve identities": that is to say, we are given an assignment F of objects of $\mathbf{D}$ to objects of $\mathbf{C}$, and an assignment, also called F, of arrows of $\mathbf{D}$ to arrows of $\mathbf{C}$, such that

$$A \xrightarrow{f} B \in \mathbf{C} \quad \text{implies } FA \xrightarrow{Ff} FB \quad \in \mathbf{D}$$

and

$$F(A \xrightarrow{f} B \xrightarrow{g} C) \;=\; FA \xrightarrow{Ff} FB \xrightarrow{Fg} FC \in \mathbf{D}.$$

In a *semi*-adjunction, we replace the usual natural bijection on homsets with a pair of maps, which need not be mutual inverses. However, we still require them to behave "naturally". The beauty of this idea lies in the following lemma.

Lemma 3.1 (Hayashi) *Suppose F, G above are in fact functors. Then a semi-adjunction between F and G yields an adjunction between F and G.*

So this concept subsumes the whole of our ordinary understanding of adjunctions, and hence all the constructions of universal gadgets defined by adjunctions[3]. In particular, in a category $\mathbf{C}$, we may define the notions of *semi-terminal object*,*semi-product* and *semi-exponential* exactly by analogy with the usual cartesian closed structure. If $\mathbf{C}$ has all the above structure, we say $\mathbf{C}$ is a *semi-cartesian closed* category, or *semi-ccc*. Hayashi's development leads to the following main results, generalising previous accounts of models of the λ-calculus with β-conversion only.

Proposition 3.1 *Semi-cccs are sound and complete for interpretations of the $\lambda\beta$-calculus.*

Proposition 3.2 *Semi-cccs can be presented algebraically.*

In [25], the second author used LEGO to prove the following theorem:

Theorem 3.1 $\mathbf{del}_1$ *is semi-cartesian closed.*

We sketch some of the ideas in the discussion below.

3.2 Identities and composition

By considering our slight modification of the underlying conversion on the terms of ECC, to remedy the failure of surjective pairing and η-conversion in the calculus, we fixed a notion of equality on arrows. With this definition, it is now trivial to establish that specifications, together with first-order deliverables as morphisms, form a category, denoted by $\mathbf{del}_1$.

The identities are given simply by

$$(s, S) \xrightarrow{(\lambda x{:}s.\ x, \lambda x{:}s.\ \lambda h{:}Sx.\ h)} (s, S) \in \ \mathbf{del}_1;$$

in LEGO, we have

```
Lego> id_del1;
value = [s|Type][S:Pred s]([x:s]x,[x|s][p:S x]p)
type  = {s|Type}{S:Pred s}del1 S S .
```

Composition is given by

$$\mathcal{S} \xrightarrow{(f,\phi)} \mathcal{T} \xrightarrow{(g,\psi)} \mathcal{U} = \mathcal{S} \xrightarrow{(\lambda x{:}s.\ g(fx), \lambda x{:}s.\ \lambda h{:}Sx.\ \psi(fx)(\phi x h))} \mathcal{U};$$

We may now verify that these definitions do indeed yield the structure of a category on $\mathbf{del}_1$[4].

3.3 Semi-Terminal object

$$Unit =_{\text{def}} (unit, \lambda u : unit.u = ()^5) : SPEC_1$$

defines a trivial specification, where we postulate the existence of a unit type in the same way as other inductive types. We then obtain, for any specification (s, S) the deliverable

$$!_{(s,S)} =_{\text{def}} (\lambda x{:}s.\ (), \lambda x{:}s.\ \lambda p{:}S\,x.\ \mathit{reflEQ}()) : \mathbf{del}_1\ S\ Unit.$$

[3]However, by contrast with adjunctions, structure defined by a semi-adjunction is *not* in general unique.

[4]Indeed, we only need the modified equality to prove the identity laws. The conversion relation is sufficient to establish associativity of composition.

[5]Here, () is the unique term of type *unit*. We write `void` in LEGO.

3.4 Binary semi-products

To obtain semi-products, we use the non-dependent Σ-type as the underlying type, with conjunction at the predicate level:

$$\mathcal{S} \times \mathcal{T} =_{\text{def}} (s \times t, \lambda p{:}s \times t.\ (S(\pi_1 p)) \wedge (T(\pi_2 p)).$$

That we indeed have a semi-product structure now follows straightforwardly from this definition.

3.5 Semi-exponentials

The notion of first-order deliverable is based on the predicate Del_1 on terms of arrow type. Precisely this predicate defines the specification which yields a semi-exponential object in $\mathbf{del}_1$. λ-abstraction and evaluation then follow from those operations in the underlying type theory on values, and implication introduction and elimination at the level of proofs.

$$\mathcal{S} \Longrightarrow \mathcal{T} =_{\text{def}} (s \to t, \lambda f{:}s \to t.\ \mathrm{Del}_1\ S\ T\ f)$$

3.6 Trivial deliverables

Every function — that is to say a term of arrow type — gives rise to a deliverable, very much in the manner of the assignment rule of Hoare logic [14] or Dijkstra's predicate transformer for assignment[9]. Namely, for

$$f : s \longrightarrow t, P : t \longrightarrow \mathit{Prop}$$

we obtain

$$(f, \lambda x{:}s.\lambda h{:}f^{\star}P.h){:}\mathbf{del}_1\ f^{\star}P\ P,\ \text{ where } f^{\star}P =_{\text{def}} \lambda x{:}s.P(fx).$$

We call these deliverables *trivial*, since they come with vacuous proofs of correctness.

3.7 A consequence rule

Logical implication induces a pointwise ordering $\subset$ on predicates, for which we have the following consequence rule, in the manner of Hoare logic:

$$\frac{\mathcal{S} \subset \mathcal{S}' \quad \mathcal{S}' \xrightarrow{(f,\phi')} \mathcal{T}' \quad \mathcal{T}' \subset \mathcal{T}}{\mathcal{S} \xrightarrow[(f,\phi)]{} \mathcal{T}}$$

In fact, this rule is subsumed by composition, since any logical entailment gives rise to a deliverable whose function component is the identity.

3.8 Pointwise construction

A basic combinator in the theory of deliverables constructs a function-proof pair from a function which returns value-proof pairs[6]. Mendler, in his thesis [28], calls such gadgets "pointwise

[6]This just corresponds to Howard's observation that, given a strong interpretation of the existential quantifier as Σ-type, the axiom of choice becomes constructively valid [15].

designs": for each argument value x, the pointwise existence of a value y (of type t) satisfying some property Tv, yields a deliverable with codomain (t, T). In detail,

$$\frac{\mathcal{F}\colon \Pi x{:}s.\ \Sigma y{:}t.\ (Sx) \longrightarrow (Ty)}{\mathcal{S} \xrightarrow{(f,\phi)} \mathcal{T}}$$

where $f =_{\text{def}} \lambda x{:}s.\ \pi_1(\mathcal{F}x)$, and $\phi =_{\text{def}} \lambda x{:}s.\ \lambda h{:}Sx.\ \pi_2(\mathcal{F}x)h$.

3.9 Inductively defined types

Provided we accept a weak definition of inductive type, it is relatively straightforward to add inductive types to the categorical structure developed so far. Categorical accounts of inductive types, via initial algebras, impose extra equalities on the iterator, due to the uniqueness clause in the definition of initial algebra. As with the semi-structures above, we only have existence, and not uniqueness, of the relevant universal arrows.

The basic idea is very simple: we add inductive types at the *Type* level, with a strong[7] elimination rule, yielding a simply typed recursor at the *Type* level, and the usual induction principle at the *Prop* level. This type is then paired with the identically true predicate. The elimination rule for first-order deliverables is easily derived, by packaging primitive recursion at the type level with induction at the predicate level. We illustrate this general idea by considering the case of natural numbers and lists.

3.9.1 Natural numbers

Recall from Section 2 that we may postulate the existence of a type of natural numbers, with two constructors, zero and successor. This yields the well-formed context

$$nat{:}\,Type,\ 0{:}nat,\ S{:}nat \longrightarrow nat$$

We typically abbreviate S to "+1" in informal mathematical language. We extend this context with a dependent elimination constant *natrecd*,

$$natrecd{:}\Pi C{:}nat \longrightarrow Type.\ \Pi z{:}C0.\ \Pi s{:}(\Pi k{:}nat.\ \Pi ih{:}Ck.\ C(k+1)).\ \Pi n{:}nat.\ Cn$$

together with reduction rules defining the δ-redexes[8] (in some context where C, z, s, n have the appropriate types):

- $natrecd\ C\ z\ s\ 0 \rhd z$;
- $natrecd\ C\ z\ s\ (n+1) \rhd s\ n\ (natrecd\ C\ z\ s\ n)$.

This is precisely expressed in LEGO as follows:

```
[nat:Type(0)];
[zero:nat];
[succ:nat -> nat];
[natrecd:{C:nat->Type}
```

[7] *Strong*, that is, because we can eliminate over *types*, and not merely propositions.
[8] *cf.* Martin-Löf type theory, or Gödel's earlier system **T** of functionals [37]. We essentially use this language of primitive recursion in all finite types as our programming language.

```
        {z:C zero}{s:{k:nat}{ih:C k}C (succ k)}{n:nat}C n];
[[n:nat][C:nat->Type][z:C zero][s:{k:nat}{ih:C k}C (succ k)]
   natrecd C z s zero ==> d
|| natrecd  C z s (succ n) ==> s n (natrecd C z s n)].
```

This yields a derived iterator and primitive recursor

$$natiter{:}\Pi\alpha{:}\,Type.\ \alpha \longrightarrow (\alpha \longrightarrow \alpha) \longrightarrow \alpha$$

and

$$natrec{:}\Pi\alpha{:}\,Type.\ \alpha \longrightarrow (nat \longrightarrow \alpha \longrightarrow \alpha) \longrightarrow \alpha$$

where[9]

$$\begin{array}{rcl} natiter\, z\ s\ 0 & =_{\mathrm{def}} & z \\ natiter\, z\ s\ (n+1) & =_{\mathrm{def}} & s\ (natiter\, z\ s\ n) \\ natrec\, z\ s\ 0 & =_{\mathrm{def}} & z \\ natrec\, z\ s\ (n+1) & =_{\mathrm{def}} & s\ n\ (natrec\, z\ s\ n), \end{array}$$

and an induction principle

$$natind : \Pi\Phi{:}nat \longrightarrow Type.\ \Pi z{:}\Phi(0).\ \Pi s{:}(\Pi k{:}nat.\ \Pi ih{:}\Phi(k).\ \Phi(k+1)).\ \Pi n{:}nat.\ \Phi n.$$

Our methodology suggests we examine derived induction principles for the iterator and recursor, since we are interested in programs, in this case of the form $natiter\, z\ s$ or $natrec\, z\ s$, together with proven propositions about them. For the iterator, we obtain

$$\frac{\Phi(z) \quad \forall y{:}\alpha.\ \Phi(y) \Longrightarrow \Phi(s\ y)}{\forall n{:}nat.\ \Phi(natiter\ z\ s\ n)}[z{:}\alpha, s{:}\alpha \longrightarrow \alpha]$$

and for the recursor

$$\frac{\Phi(z) \quad \forall k{:}nat.\ \forall y{:}\alpha.\ \Phi(k) \Longrightarrow \Phi(s\ k\ y)}{\forall n{:}nat.\ \Phi(natrec\ z\ s\ n)}[z{:}\alpha, s{:}nat \longrightarrow \alpha \longrightarrow \alpha].$$

Since we are interested in building new deliverables from less complex ones, we could of course use the dependent eliminator *natrecd* to construct terms of type $\mathbf{del}_1$, but in doing so we violate the separation of proofs from programs which distinguishes our approach. So we package recursion at the *Type* level with induction at the *Prop* level in a pair.

We introduce the predicate $Nat =_{\mathrm{def}} \lambda n{:}nat.\ true$ on the natural numbers. For each constructor of the type, we obtain a corresponding deliverable:

$$\overline{\overline{Zero =_{\mathrm{def}} (\lambda u{:}unit.\ 0,\ \lambda u{:}unit.\ \lambda h{:}\,Unit\ u.\ \top^{10}){:}\,Unit \longrightarrow Nat}}$$

$$\overline{\overline{Succ =_{\mathrm{def}} (\lambda n{:}nat.\ S\, n,\ \lambda n{:}nat.\ \lambda h{:}Nat\, n.\ \top){:}Nat \longrightarrow Nat}}$$

[9]The type α is, of course, inferred by the typechecker.

For the iterator, we obtain

$$\frac{\mathit{Unit} \xrightarrow{(z,Z)} (t,T) \quad (t,T) \xrightarrow{(s,S)} (t,T)}{\mathit{Natiter}\,(z,Z)\ (s,S){:}(\mathit{nat},\ \mathit{Nat}) \longrightarrow (t,T)}$$

and the recursor *Natrec* analogously, where the function component of $\mathit{Natiter}\ (z,Z)\ (s,S)$ (respectively *Natrec*) is $\mathit{natiter}\ (z())\ s$ (respectively *natrec*), and the proof component is obtained from the appropriate derived induction principle above.

These terms are easily obtained by refinement in LEGO. For example, here is the significant structure of *Natiter*, with the uninformative details of the proof component suppressed:

```
Lego> natiter_del1;
value = [t|Type][T|Pred t][ZZ:del1 Unit T][SS:del1 T T]
        [z=ZZ.1 void][Z=ZZ.2][s=SS.1][S=SS.2]
        (natiter z s,
         natind ([m:nat](Nat m)->T (natiter z s m)) ... )
type  = {t|Type}{T|Pred t}(del1 Unit T)->(del1 T T)->del1 Nat T
```

Example As an example of the use of these combinators, we present a correctness proof of a doubling function, given by

$$\mathit{double} =_{\mathrm{def}} \lambda n{:}\mathit{nat}.\ \mathit{natiter}\,0\ (\lambda k{:}\mathit{nat}.\ k+2)\ n.$$

Suppose we wish to show that $\mathit{double}\ n$ is even for all natural numbers n. Posed in terms of deliverables, we seek a term of type $\mathbf{del}_1\ \mathit{Nat}\ \mathit{Even}$, whose function component is *double*.

Using the rule for *Natiter* above, the problem reduces to finding:

base case $\mathit{Unit} \xrightarrow{(z,Z)} (\mathit{nat}, \mathit{Even})$. We take $z =_{\mathrm{def}} \lambda u{:}\mathit{unit}.\ 0$, and use the proof that 0 is even from Section 2;

step case $(\mathit{nat}, \mathit{Even}) \xrightarrow{(s,S)} (\mathit{nat}, \mathit{Even})$. We simply use the first-order deliverable which we constructed by composition at the end of Section 2, namely $(\lambda n{:}\mathit{nat}.\ n+2,\ \mathit{ssEE})$.

We thus obtain a non-trivial recursive first-order deliverable.

3.9.2 Lists

In much the same way as above, we may define combinators for deliverables over a type of lists. As before, we extend the context with a type constructor, in this case $\mathit{list}{:}\mathit{Type}_0 \longrightarrow \mathit{Type}_0$, together with constructors the usual *nil* and *cons*, and a dependent eliminator *listrecd*. Again we derive an iterator *listiter*, a primitive recursor *listrec* and an induction combinator *listind*.

When it comes to considering the derived induction principles for the iterator and recursor, however, we now have the freedom to specify recursions over lists of elements satisfying some

[10] Here, $\top$ is the term corresponding to *true*-introduction,

$$\top =_{\mathrm{def}} \lambda\phi{:}\mathit{Prop}.\ \lambda p{:}\phi.\ p$$

of type $\mathit{true} =_{\mathrm{def}} \Pi\phi{:}\mathit{Prop}.\ \phi \Rightarrow \phi$.

predicate, rather than over all lists of the parameter type a. That is, given some specification (a, A), we obtain a derived specification $List\,(a, A) =_{\text{def}} (list\,a,\ Listof\ A)$, where

$$\begin{aligned} Listof\ A\,(nil\ a) &=_{\text{def}} true \\ Listof\ A\,(cons\,x\,l) &=_{\text{def}} (A\,x) \wedge (Listof\ A\ l) \end{aligned}$$

defines $Listof\ A$ by primitive recursion.

We may then proceed in the same way as above, obtaining constructors

$$\frac{\quad}{Nil =_{\text{def}} (\lambda u{:}unit.\ nil\ a,\ \lambda u{:}unit.\ \lambda h{:}Unit\ u.\ \top){:}\,Unit \longrightarrow Listof\ A}$$

and

$$\frac{\quad}{Cons{:}A \longrightarrow (Listof\ A)^{Listof\ A}}$$

with function component,

$$\lambda x{:}a.\ \lambda l{:}list\,a.\ cons\,x\,l$$

and proof component

$$\lambda x{:}a.\ \lambda p{:}A\ x.\ \lambda l{:}list\,a.\ \lambda q{:}Listof\ A\ l.\ pair\,p\,q.$$

Likewise, we package together recursion and an appropriate derived induction principle, to obtain the following rules for constructing deliverables: an iterator,

$$\frac{Unit \xrightarrow{(n,N)} (t,T) \quad (a,A) \xrightarrow{(c,C)} (t,T)^{(t,T)}}{Listiter\,(n,N)\ (c,C){:}\ (list\ a,\ Listof\ A) \longrightarrow (t,T)};$$

and analogously a recursor.

4 Second-order deliverables

The system which we have described above amounts to a functional version of the well known invariants used in proofs of imperative programs. Unfortunately the specification makes no connection between the input and the output of the function. All we say is that if the input satisfies property S then the output satisfies property T, but there is no *relation* between them. Recalling the example in the introduction, we might specify that a sorting function takes lists to ordered lists, but we cannot specify that the output is a permutation of the input. The function might always produce the empty list, which is indeed sorted, but not very interesting. As a matter of fact the classical invariant proofs have the same weakness, masked by the tacit assumption that some variable which is carried through the computation does not change its value. To enforce the constraint that the output bear some relation to the input, we need to develop a compositional theory in which relations are the basic objects of study, with a notion of arrow which respects relations, rather than predicates. This gives us a categorical explanation of the idea that the pre- and postconditions are linked by having a free variahle in common, that is they are over the same context.

4.1 A thought experiment

Suppose we are given some specification $\forall x{:}s.\ \exists z{:}u.\ R(x,z)$, and we wish to find some function $f{:}s \longrightarrow u$ which satisfies it. In what sense may we refine such specifications by composition? Suppose we wish to instantiate f via the composition $f = g;h$ of two functions

$$s \xrightarrow{\quad g \quad} t \xrightarrow{\quad h \quad} u$$

where t is some intermediate type. Then, following our intuition in the case of predicates, we anticipate some intermediate specification $Q(x,y) \quad [x{:}s\ , y{:}t]$, such that g solves

$$\forall x{:}s.\ \exists y{:}t.\ Q(x,y),$$

and h solves

$$\forall x{:}s.\ (\exists y{:}t.\ Q(x,y)) \Longrightarrow \exists z{:}u.\ R(x,z).$$

This last is logically equivalent to

$$\forall x{:}s.\ \forall y{:}t.\ Q(x,y) \Longrightarrow \exists z{:}u.\ R(x,z).$$

But now we are left in something of a quandary: h, our intended solution, makes no reference to the intermediate value of y. Also, we have introduced an asymmetry between the rôles of g and h. A remedy, which underlies the definitions 4.2, and 4.3 below, is to separate the rôles of the independent parameter x and the dependent variables y, z.

We consider relations such as Q, R as the objects of study, for a fixed type s, but allowing the types t, u to vary. The following provides an appropriate notion of morphism which re-establishes a symmetry between Q and R.

Definition 4.1 An arrow from $Q(x,y) \quad [x{:}s\ , y{:}t]$ to $R(x,z) \quad [x{:}s\ , z{:}u]$ consists of the following data:

- a function $f{:}s \longrightarrow t \longrightarrow u$, that is to say a function of *two* arguments — this recovers the missing dependence we observed above;
- a proof $\phi{:}\forall x{:}s.\ \forall y{:}t.\ Q(x,y) \Longrightarrow R(x,(f\,x\,y))$.

The composition of two such arrows

$$P(x,w) \xrightarrow{(g,\psi)} Q(x,y) \xrightarrow{(h,\chi)} R(x,z)$$

is definable as

$$(\lambda x, w{:}s, r.\ h\,x\,(g\,x\,w),\ \ \lambda x, w{:}s, r.\ \lambda p{:}P(x,w).\ \psi\,x\,(g\,x\,w)\ (\phi\,x\,w\,p)).$$

We have now established a definition which respects the symmetry of source and target in our previous analysis of the decomposition of the specification $\forall x{:}s.\ \exists z{:}u.\ R(x,z)$. In so doing, we have generalised the notion of specification, and our old specification corresponds in this new setting to choosing $r =_{\text{def}} \mathit{unit}$, $P(x,w) =_{\text{def}} \mathit{true}$, and $f_{old} =_{\text{def}} \lambda x{:}s.\ f_{new}\,x\,()$.

4.2 Basic definitions

In view of the above discussion, we relativise specifications and first-order deliverables to depend on some input type s. Indeed, by observing that we may uniformly impose some condition S on the input parameter $x{:}s$, without affecting the notion of composition, we arrive at the definition of a "second-order" deliverable.

Definition 4.2 *relativised specification*
Suppose $\Gamma \vdash s : \mathit{Type}$, $\Gamma \vdash S : s \longrightarrow \mathit{Prop}$. Then a *relativised specification with respect to* (s, S) is given by a pair of terms t, R, such that

- $\Gamma \vdash t : \mathit{Type}$, and
- $\Gamma \vdash R : s \longrightarrow t \longrightarrow \mathit{Prop}$.

Definition 4.3 *second-order deliverable*
Suppose $\Gamma \vdash s : \mathit{Type}$, $\Gamma \vdash S : s \longrightarrow \mathit{Prop}$. Given two relativised specifications (t, Q) and (u, R), a *second-order deliverable over* (s, S) is a term $\mathcal{F}$ such that

$$\Gamma \vdash \mathcal{F} : \Sigma f : s \longrightarrow t \longrightarrow u. \forall x : s. S(x) \Longrightarrow \forall y : t. Q(x, y) \Longrightarrow R(x, (fxy)).$$

We define $\mathrm{Del}_2\, S\, Q\, R$ to be the predicate

$$\lambda f{:}s \longrightarrow t \longrightarrow u.\ \forall x : s. Sx \Longrightarrow \forall y : t. Q(x, y) \Longrightarrow R(x, (fxy)).$$

Definition 4.3 embodies the idea that, for each $x{:}s$ such that Sx holds, $(fx, \phi x)$ is a first-order deliverable from Qx to Rx, where $\mathcal{F} =_{\mathrm{def}} (f, \phi)$. This suggests that the study of second-order deliverables amounts to the study of first-order deliverables in an extended context. In particular we expect to obtain, for a given specification $\mathcal{S} =_{\mathrm{def}} (s, S)$, a category structure on the second-order deliverables over $\mathcal{S}$. We make a similar definition of equality on second-order deliverables to that given in Section 3 above, the details of which are left to the reader. Then we can indeed define identities and composition, given by

Identities The identity morphism from (t, Q) to itself over (s, S) is given by

$$(\lambda x{:}s.\ \lambda y{:}t.\ y,\ \ \lambda x{:}s.\ \lambda h{:}S\,x.\ \lambda y{:}t.\ \lambda q{:}Q\,x\,y.\ q)$$

Composition This is defined much as in the thought experiment above (4.1): the composition of

$$P(x, w) \xrightarrow{(f, \phi)} Q(x, y) \xrightarrow{(g, \psi)} R(x, z)$$

over (s, S) is definable as

$$(\lambda x{:}s.\ \lambda w{:}r.\ g\,x\,(f\,x\,w),\ \ \lambda x{:}s.\ \lambda h{:}S\,x.\ \lambda w{:}r.\ \lambda p{:}P\,x\,w).\ \psi\,x\,h\,(f\,x\,w)\ (\phi\,x\,h\,w\,p)).$$

This gives us a category $\mathbf{del}_2\mathcal{S}$.

If (f, ϕ) is a second-order deliverable over (s, S), we typically write

$$(t, Q) \xrightarrow{(f, \phi)} (u, R) \quad [(s, S)]\ , \text{ or even } Q \xrightarrow{(f, \phi)} R \quad [S],$$

since, as usual, the types s, t, u may be inferred by the typechecker. Our notation is intended to indicate that we are considering deliverables relative to some assumption defined by the specification (s, S). This notation is deliberately intended to echo the style of contexts in Martin-Löf type theory.

4.3 Each $\mathbf{del}_2\mathcal{S}$ is a semi-ccc

As in Section 3, we work relative to some context Γ. We have seen how a second-order deliverable (f, ϕ) over (s, S) in context Γ may be viewed as arising from a first-order deliverable in the extended context $\Gamma, x{:}s, h{:}Sx$. The conditions of definitions 4.2, and 4.3 are intended to enforce a hierarchy of dependencies in this extended context. The type t must not depend on x or h. The relation R, considered as a predicate on t in context $\Gamma, x{:}s$ does not depend on h. The function component f may not depend on h, but the proof component ϕ may do so.

Given these conditions, we may lift the structure in $\mathbf{del}_1$, by observing that the various constructions of Section 3 respect this hierarchy of dependencies. The predicates concerned need to be modified to include an explicit hypothesis Sx. We arrive at the following result.

Theorem 4.1 *For each specification $\mathcal{S}$, $\mathbf{del}_2\mathcal{S}$ has the structure of a semi-ccc.*

Proof We merely sketch some of the constructions, on the basis of the above informal intuition. Full details are in [25].

Semi-terminal object This is given by the relativised specification

$$1_S =_{\text{def}} (unit,\ \lambda x{:}s.\ \lambda u{:}unit.\ true).$$

The map $!_{(t,Q)}$, from some relativised specification (t, Q) to 1_S, has function component $\lambda x{:}s.\ \lambda y{:}t.\ ()$, and proof component

$$\lambda x{:}s. \lambda h{:}Sx. \lambda y{:}t. \lambda p{:}P\,x\,w.\ \top.$$

Semi-products Suppose we are given two relativised specifications $(t, Q), (u, R)$. Then we may form the relativised specification

$$(t, Q) \times (u, R) =_{\text{def}} (t \times u,\ \lambda x{:}s.\ \lambda p{:}t \times u.\ (Qx(\pi_1 p)) \wedge (Rx(\pi_2 p)))$$

This defines a semi-product object in $\mathbf{del}_2(s, S)$. The pairing map is given by

$$\frac{P \xrightarrow{(f,\phi)} Q \quad [S] \quad P \xrightarrow{(g,\psi)} R \quad [S]}{P \xrightarrow[(< f, g >,\ < \phi, \psi >)]{} Q \times R \quad [S]}$$

where $< f, g >$ and $< \phi, \psi >$ are appropriate pairing operations on values and proofs respectively. Projections are similarly straightforward to define.

Semi-exponentials Just as the predicate Del_1 defined a semi-exponential object in the category $\mathbf{del}_1$, we may define a semi-exponential object in $\mathbf{del}_2\ (s, S)$, using the relativised specification to which Del_2 gives rise. More precisely, suppose $(r, P), (t, Q), (u, R)$ are relativised specifications. We obtain a relativised specification

$$R^Q =_{\text{def}} (t \longrightarrow u,\ \lambda x{:}s.\ \lambda f{:}t \longrightarrow u.\ \forall y{:}t. Q\,x\,y \Longrightarrow R\,x\,(fy)).$$

If $\Gamma \vdash f : s \longrightarrow t \longrightarrow u$, and $\Gamma \vdash \phi : \mathrm{Del}_2\,S\,Q\,R\,f$, then $\Gamma, x{:}s \vdash \phi x : R^Q\,fx$. Moreover, given

$$P \times Q \xrightarrow{\mathcal{F}} R \quad [S]$$

we obtain

$$P \xrightarrow{\Lambda(\mathcal{F})} R^Q \quad [S]$$

by currying in the obvious way: if $\mathcal{F} =_{\text{def}} (f, \phi)$, then $\Lambda(\mathcal{F}) =_{\text{def}} (\hat{f}, \hat{\phi})$, where $\hat{f}$ and $\hat{\phi}$ are appropriate currying operations on values and proofs respectively. We may similarly define an evaluation map, details of which are left to the imaginative reader: it is rather less taxing to develop this construction by refinement in LEGO. Likewise, the proofs that these data meet Hayashi's conditions for a semi-exponential are best dealt with by refinement.

■

4.4 $\mathbf{del}_2$: an indexed category over $\mathbf{del}_1$

The categorically minded reader now asks herself what relationships exist between the various categories $\{\mathbf{del}_2\mathcal{S} | \mathcal{S} \in SPEC_1\}$, and to what extent we may elaborate upon the structure of this collection. In particular, she may ask what is the relationship between $\mathbf{del}_2\mathcal{S}$ and $\mathbf{del}_2\mathcal{T}$, given a first-order deliverable from $\mathcal{S}$ to $\mathcal{T}$. A moment's pause should convince her that composition in $\mathbf{del}_1$ should induce an operation on second-order deliverables, since they somehow are no more than first-order deliverables, except that they are defined in an extended context. In other words, we are groping towards the following theorem:

Theorem 4.2 $\mathbf{del}_2$ *is an indexed category [19, 1] over* $\mathbf{del}_1$*, whose fibres are semi-cccs, with semi-cc structure strictly preserved by reindexing along arrows in* $\mathbf{del}_1$*.*

4.5 Pullback functors

The above theorem depends on the existence of pullback functors, which translate, or reindex, data between the categories $\mathbf{del}_2\mathcal{S}$. The obvious definition works, and moreover trivially respects the equality of objects and arrows, so we do indeed have pullback functors — and they are functors, not merely semi-functors, since identities and composition are preserved. It is then a straightforward, and tedious, task to verify that these operations compose, and strictly preserve the structure in each fibre.

Definition 4.4 *pullback along a first-order deliverable*
Suppose we are given specifications $\mathcal{S}, \mathcal{T}$, and a first-order deliverable $\mathcal{S} \xrightarrow[\mathcal{T}]{(k,K)}$. We define an operation of *pullback along* (k, K), where we abuse notation in the standard way by employing the same symbol for the operation on objects and arrows, as follows: given a relativised specification $\mathcal{Q} =_{\text{def}} (u, Q)$ with respect to $\mathcal{T}$, let

$$(k, K)^{\star}\mathcal{Q} =_{\text{def}} \lambda x{:}s.\ \lambda z{:}u.\ Q\,(kx)\,z;$$

moreover, given a relativised specification $\mathcal{R} =_{\text{def}} (v, R)$, and a second-order deliverable

$$\mathcal{Q} \xrightarrow{(f, \phi)} \mathcal{R} \quad [\mathcal{T}]$$

we define $(k, K)^{\star}(f, \phi)$ to be the pair

$$(\lambda x{:}s.\ \lambda z{:}u.\ f\,(k\,x)\,z,\ \lambda x{:}s.\ \lambda h{:}Sx.\ \lambda z{:}u.\ \lambda p{:}\mathcal{Q}\,(k\,x)\,z.\ \phi\,(k\,x)\,(K\,x\,h)\,z\,p).$$

Lemma 4.1 *$(k,K)^{\star}\mathcal{Q}$ is a relativised specification with respect to $\mathcal{S}$. Moreover, $(k,K)^{\star}(f,\phi)$ is a second-order deliverable from $(k,K)^{\star}\mathcal{Q}$ to $(k,K)^{\star}\mathcal{R}$.*

Lemma 4.2 *$(k,K)^{\star}$ preserves identities and composition.*

So, indeed, we do have the the existence of functors between the fibres $\mathbf{del}_2$. That they are a satisfactory notion of reindexing requires us to show that they obey the condition $\mathcal{H}^{\star};\mathcal{K}^{\star} \cong (\mathcal{K};\mathcal{H})^{\star}$. In fact, more is true. We have the following:

Lemma 4.3 *The reindexing is* strict*, in the sense that*

$$\mathcal{H}^{\star};\mathcal{K}^{\star} = (\mathcal{K};\mathcal{H})^{\star}.$$

We now turn to the remainder of Theorem 4.2, namely that the pullback functors preserve the structure of a semi-ccc in each fibre. As above, we find that the structure is preserved strictly. We examine only the case of exponentials, the cases of products and terminal object being exactly similar, and rather easier.

Lemma 4.4 *In the notation of Theorem 4.1 above, with $\mathcal{V} \xrightarrow[\mathcal{S}]{\mathcal{K}}$, we have*

$$\mathcal{K}^{\star}(R^{Q}) = (\mathcal{K}^{\star}R)^{\mathcal{K}^{\star}Q}.$$

Proposition 4.1 *Suppose $(r,P),(t,Q),(u,R)$ are relativised specifications with respect to $\mathcal{S}$. Given*

$$P \times Q \xrightarrow{\mathcal{F}} R \quad [S] \quad \textit{and} \quad \mathcal{V} \xrightarrow{\mathcal{K}} \mathcal{S}$$

we have $\Lambda(\mathcal{K}^{\star}\mathcal{F}) \equiv \mathcal{K}^{\star}\!*(\mathcal{F})$.

4.6 Second-order deliverables for natural numbers and lists

In the context of second-order deliverables, the situation regarding inductive types is less well understood. We do not regard this section as giving a definitive account, but the examples we have considered suggest that we have a usable set of combinators for reasoning about recursive programs.

We take as our guiding motivation the derived induction principles of the last section. Since we now work in the relativised case, these will be subtly altered by the presence of the induction variable.

This means, for the case of natural numbers, that we now examine proofs of statements of the form[11]:

$$\forall n{:}nat.\ R\ n\ (natrec\ z\ s\ n)$$

where, for some type t, $z{:}t$ and $s{:}nat \longrightarrow t \longrightarrow t$. A proof of this, by induction, yields

$$R\ 0\ z \quad \text{and} \quad \forall k{:}nat.\ \forall y{:}t.\ R\ k\ y \Longrightarrow R\ (k+1)\ (s\ k\ y)$$

as the requisite hypotheses in the base and step cases. We may now recognise the second hypothesis as the logical component of some second-order deliverable, whose function component is s.

[11] We only consider *natrec*, since *natiter* is a degenerate instance of it.

The question arises as to how to view the first hypothesis $R\ 0\ z$. Do we regard it as part of some first or second-order deliverable? In a sense, neither, in the choice we have made in the current version of deliverables. If we examine the derived rule of induction again, but this time rephrased as

$$\frac{\forall k{:}nat.\ \forall y{:}t.\ R\ k\ y \Longrightarrow R\ (k+1)\ (s\ k\ y)}{\forall n{:}nat.\ \forall z{:}t.\ R\ 0\ z \Longrightarrow R\ n\ (natrec\ z\ s\ n)}$$

this isolates how we currently view recursions at the second-order level. Namely, we see the function which recursively applies s to an *arbitrary* initial value z as the function component of some second-order deliverable, whose proof component is the proof by induction of the conclusion

$$\forall n{:}nat.\ \forall z{:}t.\ R\ 0\ z \Longrightarrow R\ n\ (natrec\ z\ s\ n).$$

As observed above, the hypothesis in the step case of induction arises as the proof component of a second-order deliverable

$$R \xrightarrow{(s,S)} (+1)^{\star}R \quad [1_{nat}]$$

where $(+1)^{\star}R$, otherwise written $R[n+1/n]$, is the relation

$$\lambda n{:}nat.\ \lambda y{:}t.\ R\ (n+1)\ y.$$

In like manner, we write $0^{\star}R$, or $R[0/n]$, for the relation

$$\lambda n{:}nat.\ \lambda y{:}t.\ R\ 0\ y.$$

We thus obtain the second-order deliverable constructor for *nat* recursions as the following derived rule

$$\frac{R \xrightarrow{(s,S)} (+1)^{\star}R \quad [1_{nat}]}{Natrec_2\ (s,S){:}0^{\star}R \rightarrow R \quad [1_{nat}]}$$

where $Natrec_2$ has function component

$$\lambda n{:}nat.\ \lambda z{:}t.\ natrec\ z\ s\ n.$$

The principle reason for making this choice of representation is a pragmatic one, based partly on experience, and on the behaviour of unification in the typechecker. If we were to mimic the construction of first-order deliverables by induction, we would expect some rule with one hypothesis for each constructor of the datatype, such as for example

$$\frac{1 \xrightarrow{(z,Z)} 0^{\star}R \quad [1_{nat}] \qquad R \xrightarrow{(s,S)} (+1)^{\star}R \quad [1_{nat}]}{Natrec_2'\ (z,Z)\ (s,S){:}1 \rightarrow R \quad [1]}.$$

We would typically apply such a rule in a top-down proof, to a subgoal of the form
`del2 ?n ?m R`
In a top-down development, where we may construct deliverables using all the constructions described above, we would like the instantiation of `?m` to be both as general as possible, to allow for subsequent development, and yet to allow unification to constrain `?m` to make the application valid. Our choice of the above rule for $Natrec_2$ seems to achieve this. We do not regard this choice as necessarily definitive, however: it merely represents our present view.

4.7 Lists

We may extend this analysis to the case of lists, where, as in the case of first-order deliverables, we find the richer structure of lists reflected in a richer collection of predicates and relations.

Firstly, if we work in the fibre $\mathbf{del}_2\ 1_{list\ a}$, then we obtain in exactly the same way as above, the following derived rule:

$$\frac{\Pi x{:}a.\ R \xrightarrow{\mathcal{F}} (cons\ x)^{\star}R \quad [1_{list\ a}]}{Listrec_2\ \mathcal{F}{:}(nil)^{\star}R \rightarrow R \quad [1_{list\ a}]}$$

where

$$(cons\ x)^{\star}R =_{\mathrm{def}} \lambda l{:}list\ a.\ \lambda y{:}t.\ R\ (cons\ x\ l)\ y$$

and

$$(nil)^{\star}R =_{\mathrm{def}} \lambda l{:}list\ a.\ \lambda y{:}t.\ R\ (nil\ a)\ y.$$

But already in this rule we find something new: the outermost Π binding. That is to say, the rule has as its premise a *dependent* family of second-order deliverables. This phenomenon arises from the parameter type a of the lists in question. The rule is susceptible to the same criticisms as the rule for $Natrec_2$ above, but also the criticism that we have accorded a different status to the parameter type. In particular, it does not seem to be constrained by any predicate A we might impose on a. Our justification, as above, is essentially pragmatic. We have found this rule to be a useful construction, as in the example of minimum finding in a list.

We can obtain forms of this rule in which the input list is further constrained. We may, for example, consider the predicate $Listof\ A$, for some predicate A on a. In fact, since we are now considering second-order deliverables, where we can take into account relations which depend on both the input variable and the result of some computation step, we may extend this predicate to a dependent version, which we call $depListof\ A$, defined as follows:

$$\begin{aligned} depListof\ \Phi\ (nil\ a) &=_{\mathrm{def}} true \\ depListof\ \Phi\ (cons\ x\ l) &=_{\mathrm{def}} (\Phi\ x\ l) \wedge (depListof\ \Phi\ l) \end{aligned}$$

Here Φ is some *relation* between values of the variable x varying over the parameter type a, and lists over a. An example is the predicate $Sorted$, for which we take $\Phi\ x\ l =_{\mathrm{def}} x \preceq l$, the relation that x is less than each element of the list l. This crops up in the example of an insert sort in the next section.

This introduces an extra hypothesis into the induction scheme we must consider. Suppose we wish to prove

$$\forall l{:}list\, a.\ (depListof\ \Phi\ l) \Longrightarrow R\ l\ (listrec\, n\ c\ l)$$

where $n{:}t$, $c{:}a \longrightarrow t \longrightarrow t$. A proof by induction generates the following hypotheses for each constructor.

base case $true \Longrightarrow R\, nil\ n$, which reduces logically to $R\, nil\ n$. As with the rules for $Natrec_2$, we shall fold this assumption into the rule as the initial relation in the second-order deliverable we eventually derive.

step case Formally, we obtain

$$\forall x{:}a.\ \forall l{:}list\, a.\ ((depListof\ \Phi\ l) \Rightarrow R\ l\ (listrec\, n\ c\ l)) \Longrightarrow$$

$$((depListof\ \Phi\ (x :: l)) \Rightarrow R\ (x :: l)\ (c\ x\ l\ (listrec\ n\ c\ l))).$$

Two simplifications present themselves. The first is to replace the explicit mention of $(listrec\ n\ c\ l)$ by an additional universally quantified parameter y. The second is to observe that $depListof\ \Phi\ (cons\ x\ l) \Longrightarrow depListof\ \Phi\ l$. Combining these, we obtain as an induction hypothesis in the step case

$$\forall x{:}a.\ \forall l{:}list\ a.\ \forall y{:}t.\ (depListof\ \Phi\ (x :: l)) \Longrightarrow R\ l\ y \Longrightarrow R\ (x :: l)\ (c\ x\ l\ y).$$

In this form, we see the logical part of a second-order deliverable emerge.

We thus obtain the following derived rule, which yields a second-order deliverable with function component *listrec* from a dependent family of second-order deliverables:

$$\frac{\mathcal{F}{:}\Pi x{:}a.\ R \ \rightharpoonup\ (cons\ x)^{\star}R \quad [(cons\ x)^{\star}depListof\ \Phi]}{depListrec_2\ \mathcal{F}{:}nil^{\star}R \ \rightharpoonup\ R \quad [depListof\ \Phi]}$$

5 Examples

In his thesis [25], the second author considered a number of example developments using the methodology outlined above, culminating in a machine-checked proof of the Chinese remainder theorem. This last example is too long to include here, although it perhaps best exhibits some of the strengths of our approach. We hope to discuss it in a forthcoming paper.

To illustrate the discussion of the previous sections, we give two examples of the use of deliverables in small-scale program development. The use of deliverables may seem heavy-handed and even counter-productive for these smaller examples. The discussion is taken from [25], where detailed accounts in LEGO of the derivations may be found.

5.1 Finding the minimum of a list

An example which is treated several times in the literature [30, 35]. In our case, it involves a somewhat unnatural reperesentation, which makes non-trivial use of the semi-cartesian closed structure in the fibres $\mathbf{del}_2$.

5.2 The mathematical specification

Suppose R is a decidable total order on some type α, which for the purposes of this example contains a maximal element a_0 (this avoids having to consider exceptions for the case of the *nil* list). We distinguish between R, a *boolean* valued function, and the *relation* $\lambda a, b{:}\alpha.\ R\,a\,b = tt$, denoted $\leq_R$. Then we may specify the minimum of a list as follows:

$$\forall l{:}list\alpha. l \neq nil \Rightarrow \exists m{:}\alpha. m \in l \wedge (\forall a{:}\alpha. a \in l \Rightarrow m \leq_R a).$$

We abbreviate the second conjunct to $m \preceq_R l$. We may easily express a solution to this specification in SML as follows (:: adds an element to a list):

```
fun min a b = if (R a b) then a else b;

fun minelemaux nil = (fn a => a) |
    minelemaux (b::l) = (fn a => (min a (minelemaux l b));

fun minelem nil = a_0 |
    minelem (a::l) = minelemaux l a;
```

We have explicitly curried the function definition of [35][12]

```
fun minelem (a::nil) = a
 |  minelem(a::b::l) = min a (minelem (b::l));
```

since the type system of ECC is too strict to allow such a definition based on pattern matching. Our definition is by recursion on the first argument l of `minelemaux`; the corresponding proof is by induction on l. We seek to verify that `minelemaux` meets the following specification:

$$\forall l{:}list\,\alpha.\ \exists f{:}\alpha \to \alpha.\ \forall a{:}\alpha.\ fa \in a :: l \wedge fa \preceq_R a :: l;$$

The proof for `minelem` follows by composition with a (suitably relativised) deliverable for application.

5.3 The correctness proof

We just consider the verification of `minelemaux`. As indicated, the proof is by induction.

Base case

$$\exists f{:}\alpha \to \alpha.\ \forall a{:}\alpha.\ fa \in a :: nil \wedge fa \preceq_R a :: nil.$$

The condition $fa \in a :: nil$ forces us to choose $f =_{\text{def}} \lambda a{:}\alpha.\ a$. For a reflexive R, the second conjunct is then satisfied.

Step case Suppose

$$\exists f{:}\alpha \to \alpha.\ \forall a{:}\alpha.\ fa \in a :: k \wedge fa \preceq_R a :: k.$$

Then for $b \in \alpha$, taking $g =_{\text{def}} \lambda a{:}\alpha.\ min\ a\ (f\,b)$, we obtain for all $a \in \alpha$, by cases on $R\ a\ (f\,b)$:

- $R\ a\ (f\,b) = true$, and hence $min\ a\ (f\,b) = a$. Now $a \in a :: b :: k$, and $a \preceq_R a :: b :: k$, since $a \leq_R a$, and $a \leq_R (f\,b) \preceq_R b :: k$, by hypothesis, and the transitivity of $\leq_R$.
- $R\ a\ (f\,b) = false$, and hence $min\ a\ (f\,b) = f\,b$. We have $fb \in b :: k$, by hypothesis, and hence $fb \in a :: b :: k$. Again, by hypothesis, $f\,b \preceq_R b :: k$, and $fb \leq_R a$. Hence $fb \preceq_R a :: b :: k$, and we are done.

5.4 The development in terms of deliverables

We give an outline of the proof in the style of a derivation tree, indicating the combinators used to resolve significant subgoals.

Let

$$MinAuxSpec =_{\text{def}} \lambda l{:}list\alpha.\ \lambda f{:}\alpha \longrightarrow \alpha.\ \forall a{:}\alpha.\ fa \in a :: l \wedge fa \preceq_R a :: l.$$

Abbreviating the trivial specifications $\lambda l{:}list\alpha.\ true$ and $\lambda l{:}list\alpha.\ \lambda u{:}unit.\ true$ to 1 (since they are terminal objects, respectively in $\mathbf{del}_1$, and $\mathbf{del}_2\ (list\alpha,\ 1)$), we seek

$$1 \xrightarrow{minelemaux} MinAuxSpec \quad [1]$$

[12] Sannella in fact treats the *maximum* of the list.

We use composition to enable us to exploit our recursion combinator for second-order deliverables over lists of Section 4.6, to package up the proof by induction:

$$\frac{1 \xrightarrow{base} ?1 \quad [1] \quad ?1 \xrightarrow{step} MinAuxSpec \quad [1]}{1 \xrightarrow{minelemaux} MinAuxSpec \quad [1]}(compose)$$

and

$$\frac{\Pi a{:}\alpha.\ MinAuxSpec \xrightarrow{\mathcal{F}_{step}} (cons\ a)^{\star}MinAuxSpec \quad [1_{list\ \alpha}]}{?1 \xrightarrow{step} MinAuxSpec \quad [1]}(Listrec_2)$$

which instantiates subgoal ?1 with the specification $nil^{\star}MinAuxSpec$.

We resolve the base case of our induction using a pointwise construction, and thereafter exactly the informal reasoning above. This is sufficient to allow us to completely close this branch of the derivation.

$$\frac{\begin{array}{l} value = \lambda a{:}\alpha.\ a \\ proof : a \in [a] \wedge a \preceq_R [a] \end{array}}{1 \xrightarrow{base} nil^{\star}MinAuxSpec \quad [1]}(pointwise)$$

This leaves the step case. We extend the context with a free variable b of type α, and then conclude with another pointwise construction. The proof follows the informal argument above.

$$\frac{\begin{array}{l} value = \lambda a{:}\alpha.\ min\ a\ (f\ b) \\ proof : \ldots \end{array}}{MinAuxSpec \xrightarrow{\mathcal{F}_{step}\ b} (cons\ b)^{\star}MinAuxSpec \quad [1]}(pointwise)$$

5.5 Insert sort

We chose insert sort, since it is expressible naturally within primitive recursion, as opposed to more efficient algorithms such as Hoare's quicksort, which has a natural expression only in terms of general recursion.

5.6 The mathematical specification

Informally, this is straightforward enough. For every list l, over some type α which carries a decidable linear ordering, there exists a sorted list m which is a permutation of l. In a formal treatment, we must give explicit representations of the notions of "sortedness" and "permutation". We used an impredicative definition of the permutation relation, which we do not discuss here. The predicate $Sorted$ was defined by recursion:

$$\frac{}{Sorted(nil)} \qquad \frac{Sorted\,(l) \quad a \preceq l}{Sorted\,(a :: l)}\ [a{:}\alpha, l{:}list\ \alpha]$$

where

$$\frac{}{a \preceq nil \quad [a{:}\alpha]} \qquad \frac{a \preceq l \quad a \leq b}{a \preceq b :: l}\ [a, b{:}\alpha, l{:}list\ \alpha]$$

As we observed in the section on dependent list recursion, *Sorted* may be seen as an instance of the *depListof* constructor, while the predicate $\lambda l{:}list\,\alpha.\ a \preceq l$, for $a{:}\alpha$, is just an instance of *Listof*.

5.7 A correctness proof for insert sort

Here is the ML prototype for the sort that we wrote, with a view to proving correct:

```
fun listrec n c l = fold ( fn (a,m) => c a m) l n;
val swapcons = fn b => (fn p =>
                        let val (c,m) = p
                         in  (max(c,b), (min(c,b):: m))
                          end);
fun scons a l = let val (b,m) = listrec (a,nil) swapcons  l
                in b::m
                 end;
val sort = listrec nil scons;
```

and here is its translation into LEGO code:

```
[swapcons = [A|Type][b:A][p:A # (list A)]
            ((min p.1 p.2),cons (max p.1 p.2) m)];

[scons = [A|Type][a:A][l:list A]
         [p = listrec (a,nil) swapcons  l](cons p.1 p.2)];

[sort = listrec nil scons];
```

We anticipate two levels of induction in the correctness proof for this function, since sort is defined by nested recursion. We recall the derived induction principles for recursive program phrases which we used to explain recursive deliverables:

$$\text{for } \phi{:}(list\ \alpha) \longrightarrow \beta \longrightarrow Prop, \quad n{:}\beta, \quad c{:}\alpha \longrightarrow (list\ \alpha) \longrightarrow \beta \longrightarrow \beta$$

$$\dfrac{\phi\ nil\ n \qquad \begin{array}{c}\left\lceil \phi\ l\ b\ \ [a{:}\alpha,\ l{:}list\ \alpha,\ b{:}\beta] \right\rceil \\ \vdots \\ \phi\ \ (a{::}l)\ \ (c\ a\ l\ b)\end{array}}{\forall l{:}list\ \alpha.\phi\ l\ (listrec\,n\ c\ l)}$$

a consequence of which is the following induction principle for sorted lists, which is just the proof component of an instance of the $depListrec_2$ combinator we considered above:

$$\text{for } \phi{:}(list\ \alpha) \longrightarrow \beta \longrightarrow Prop, \quad n{:}\beta, \quad c{:}\alpha \longrightarrow (list\ \alpha) \longrightarrow \beta \longrightarrow \beta$$

$$\dfrac{\phi\ nil\ n \qquad \begin{array}{c}\left\lceil Sorted\,(a{::}l),\ \ \phi\ l\ b\ \ [a{:}\alpha,\ l{:}list\ \alpha,\ b{:}\beta] \right\rceil \\ \vdots \\ \phi\ \ (a{::}l)\ \ (c\ a\ l\ b)\end{array}}{\forall l{:}list\ \alpha.\ \ Sorted\ \ l \Longrightarrow \phi\ l\ (listrec\,n\ c\ l)}\ \text{Sorted List Induction}$$

This induction principle exemplifies the motivation for second-order deliverables: namely that a relation ϕ between lists holds, *only* on the condition that one of them is sorted.

The correctness proof proceeds by application of the first induction principle, for the outermost recursion: this produces as subgoals

(a) $\mathit{Sorted}(\mathit{nil}),\ \mathit{nil} \sim \mathit{nil}$

(b) $\forall a : \alpha.\forall l, m{:}\mathit{list}\ \alpha.\ (\mathit{Sorted}\,(m) \wedge l \sim m) \Longrightarrow$
$(\mathit{Sorted}\,(\text{scons}\ a\ m) \wedge a :: l \sim (\text{scons}\ a\ m))$.

Now (a) is immediate, and (b) reduces to

$$\forall a : \alpha.\forall m{:}\mathit{list}\ \alpha.\mathit{Sorted}\,(m) \Longrightarrow (\mathit{Sorted}\,(\text{scons}\ a\ m) \wedge a :: m \sim (\text{scons}\ a\ m))$$

$\forall$-introduction extends the context with the assumption $[a{:}\alpha]$, and in this extended context we define the relativised specification

$$\psi_a \equiv \lambda m, n{:}\mathit{list}\ \alpha.\mathit{Sorted}\,(n) \wedge a :: m \sim n.$$

Now we may apply Sorted List Induction to yield the following subgoals in the new context:

(c) $a :: \mathit{nil} \sim a :: \mathit{nil} \wedge \mathit{Sorted}\,(a :: \mathit{nil})$

(d) $\forall b : \alpha.\forall m, n{:}\mathit{list}\ \alpha.\ \mathit{Sorted}\,(b :: m) \wedge \psi_a mn \Longrightarrow \psi_a(b :: m)(\text{swapcons}\ b\ m)$.

(c) is trivial, and (d) reduces to

$$\forall b, c : \alpha.\forall m, n{:}\mathit{list}\ \alpha.\mathit{Sorted}\,(b :: m) \wedge \mathit{Sorted}\,(c :: n) \wedge a :: m \sim c :: n \Longrightarrow$$

$$\mathit{Sorted}\,(\min\,(b,c) :: \max\,(b,c) :: n) \wedge a :: b :: m \sim \min\,(b,c) :: \max\,(b,c) :: n$$

This last goal rests, apart from some trivial properties of $\sim$, on the following:

Lemma 5.1 *Suppose $a, b, c \in \alpha$, and $m, n \in \mathit{list}\ \alpha$ such that:*

- $\mathit{Sorted}\,(c :: n)$*;*
- $c :: n \sim a :: m$*;*
- $b \preceq m$.

Then $\max\,(b,c) \preceq n$.

5.8 The proof recast in terms of deliverables

We have a double recursion in the definition of our program, which translates into a nested list recursion at the level of deliverables. The base case of each induction is resolved with a simple pointwise construction. So too is the inner step case for the verification of the `swapcons` function, with some additional propositional reasoning.

However, there is a point of considerable delicacy in the course of this proof. There is a stage at which we must shift from a second-order deliverable over 1, the identically true predicate on lists, defined by the outermost recursion, to the inner recursion on second-order deliverables defined over the predicate *Sorted*. It appears that we must eliminate the *input* parameter l using the permutation relation, which moves the condition $\mathit{Sorted}(m)$ from being a condition on a *result* to being a condition on the *input* to *scons*. This is the reduction in case (b) above, from

$$\forall a:\alpha.\forall l, m{:}list\ \alpha.(Sorted\,(m) \wedge l \sim m) \Longrightarrow (Sorted\,(\mathrm{scons}\ a\ m) \wedge a :: l \sim (\mathrm{scons}\ a\ m)$$

to

$$\forall a:\alpha.\forall m{:}list\ \alpha.Sorted\,(m) \Longrightarrow (Sorted\,(\mathrm{scons}\ a\ m) \wedge a :: m \sim (\mathrm{scons}\ a\ m)$$

This shift allows us to apply the principle of Sorted List Induction.

This is rather inconvenient, and moreover it seems that only *ad hoc* solutions exist to resolve the difficulty. Rather than attempt a detailed exposition here, we merely remark on the rather unsatisfactory nature of this step in our derivation. The details are in [25].

We outline the significant refinement steps in the derivation.

We seek, as a top-level goal

$$1 \xrightarrow{insertsort} SortSpec\ \ [1]$$

where $SortSpec\ l\ m =_{\mathrm{def}} Sorted(m) \wedge\ l \sim m$.

The outermost recursion uses the $Listrec_2$ combinator, which we apply after splitting the initial goal with the composition operator.

$$\frac{1 \xrightarrow{base} ?1\ \ [1] \quad ?1 \xrightarrow{step} SortSpec\ \ [1]}{1 \xrightarrow{insertsort} SortSpec\ \ [1]}(compose)$$

$$\frac{\Pi a{:}\alpha.\ SortSpec \xrightarrow{\mathcal{F}step} (cons\ a)^{\star}SortSpec\ \ [1_{list\ \alpha}]}{?1 \xrightarrow{step} SortSpec\ \ [1]}(Listrec_2)$$

instantiates subgoal ?1 with the specification $nil^{\star}SortSpec$.

The base case of this recursion/induction is resolved by a pointwise construction. Moreover, we can prove that any permutation of the *nil* list must be equal to *nil*, and the *nil* list is trivially sorted. Hence we obtain the correct instantiation

$$\frac{\begin{array}{l} value = nil \\ proof : Sorted(nil) \wedge nil \sim nil \end{array}}{1 \xrightarrow{base} nil^{\star}SortSpec\ \ [1]}(pointwise)$$

The step case is a little more complicated. We first introduce the parameter $a:\alpha$ in the $Listrec_2$ rule. Recall that the algorithm we considered above contains a `let` construct.

```
fun scons a l = let val (b,m) = listrec (a,nil) swapcons  l
                in b::m
                 end;
```

Now composition of deliverables explicates the `let` construct. For the second component of this composition, we just use the second-order deliverable analogue of the trivial construction of Section 3. We then fold this function into a relativised specification, with which we pursue the inner recursion/induction:

$$\dfrac{SortSpec \rightharpoonup SortSpec' \quad [1] \qquad \dfrac{function = \lambda p{:}\alpha \times list\alpha.\ \pi_1(p) :: \pi_2(p)}{SortSpec' \rightharpoonup (cons\ a)^{\star}SortSpec \quad [1]}(functional)}{SortSpec \xrightarrow{\mathcal{F}_{step}\ a} cons\ a^{\star}SortSpec \quad [1]}(compose)$$

where $SortSpec' =_{\text{def}} (\lambda p{:}\alpha \times list\alpha.\ \pi_1(p) :: \pi_2(p))^{\star}(cons\ a)^{\star}SortSpec$

Eliding the details, we massage the goal $SortSpec \rightharpoonup SortSpec' \quad [1]$ into a form in which we may exploit the principle of Sorted List Induction. It turns out that the target specification remains as $SortSpec'$:

$$\dfrac{\dfrac{1 \rightharpoonup SortSpec' \quad [Sorted]}{\dots}}{SortSpec \rightharpoonup SortSpec' \quad [1]}$$

As indicated above, we are now in a position to use the $depListrec_2$ combinator. Again, we preface its application with an appeal to composition.

$$\dfrac{1 \xrightarrow{base'} ? \quad [Sorted] \quad ? \xrightarrow{step'} SortSpec \quad [Sorted]}{1 \rightharpoonup SortSpec \quad [Sorted]}(compose)$$

$$\dfrac{\Pi b{:}\alpha.\ SortSpec' \xrightarrow{\mathcal{G}_{step'}} (cons\ b)^{\star}SortSpec' \quad [(cons\ b)^{\star}Sorted]}{? \rightharpoonup SortSpec' \quad [Sorted]}(depListrec_2)$$

endlegolist

The proof concludes by considering a pointwise construction in the base case of the induction,

$$\dfrac{\begin{array}{l} value = (a, nil) \\ proof : Sorted([a]) \wedge [a] \sim [a] \end{array}}{1 \xrightarrow{base'} nil^{\star}(SortSpec') \quad [Sorted]}(pointwise)$$

and then an account of the verification of the `swapcons` function in the $step'$ case. We omit the details, which exploit the informal argument of Lemma 5.1.

6 A set-theoretic model

In the foregoing discussion, our emphasis lay on using and constructing deliverables in the context of a particular type theory and its machine implementation. However, the idea of deliverables need not be restricted to this particular seeting. We observed that one could model deliverables in the framework of elementary set theory. The thesis [25] gives a detailed categorical account of this model, showing how it supports the interpretation of a subtheory of ECC. Indeed, the model is parametric in the choice of an underlying *topos*(categorical model of set theory). We sketch the main ideas. We refer the reader to [25] for a complete account.

In this interpretation, a (closed) type is given by a pair of sets (A, A'), with $A' \subseteq A$. In particular, the type *Prop* of propositions, is given by the pair (Ω, Ω), where Ω is the subobject classifier. A morphism between such objects is given by the obvious abstract notion of first-order deliverable, i.e. an arrow on the underlying sets which respects the distinguished subsets. This data gives us an interpretation of contexts and morphisms between them.

Types in a given context $\Gamma = (X_0, X_1)$ are given by the relativised specifications, i.e. pairs of sets (A, R), where now $R \subseteq X_0 \times A$. Without going into further detail, this structure gives rise to a *fibration*, whose fibres essentially consist of the second-order deliverables described above. Moreover, this fibration has sums, products and a small subfibration and thus supports an interpretation of the "$Prop{:}\,Type_0{:}\,Type_1$" fragment of ECC. In the presence of set-theoretic universes, given for example by inaccesible cardinals, we may model the whole of ECC.

7 Related work

Two approaches to a logical account of formal program development are well known. The first relies on annotating programs with logical formulae, and expressing the correctness of a program in terms of logical deductions. This has its origins in the work of Floyd, Hoare and others in the 'Sixties [14, for example]. The idea of a deliverable clearly has echoes of this idea, but brought into the functional setting. It also avoids the defect of the Floyd/Hoare style in having object language and meta-logical variables on the same footing as object variables of our chosen type theory.

The second approach, based on various intuitionistic type theories, has been to develop constructive proofs, and use realisability techniques to extract algorithmic information. Both Martin-Löf type theory [5, 29] and the Calculus of Constructions [6, 30, 31] have been used in this style. Our work uses ideas from both these schools. Most influential has been the theory of subsets in Martin-Löf type theory. This has been given an eloquent treatment in [29, Chapter 18], to which the reader is referred for a detailed discussion. The central problem in using constructive proofs as a programming discipline is that proofs contain redundant information. Dependent types allow us to express logical predicates as types, but do not permit the representation of any more recursive functions. For the Calculus of Constructions, this has a precise statement in the result of Berardi and Mohring [30, 2]:
Theorem *$\mathcal{CC}$ is conservative over F_ω.*
Consequently, $\mathcal{CC}$ can represent, in the standard (Church) representation of functions on inductive types, no more functions than F_ω. The proof is based on a syntactic mapping, the so-called Berardi-Mohring projection. Mohring used this mapping as an extraction function, which, coupled with the associated realisability predicate, allows a powerful and flexible approach to program development from proofs. Under this interpretation, an arbitrary type is interpreted as a type, together with a predicate (the realisability predicate) defined over it.

The approach taken in [29] is to separate computationally relevant proofs from the purely logical, via a translation of the judgments of the basic formal system into multiple judgments. This translation permits the formation of a "subset type" $\{x \in A | B(x)\}$, for which a reasonable elimination rule may be given. An attempt to equip the basic theory with such a type (which may be used to precisely hide the information of the precise nature of a proof that predicate $B(x)$ holds) yields very unsatisfactory results [33, 34, 29].

Given the basic theory of types and terms in Martin-Löf type theory, the first step is to extend the theory with a notion of *proposition* and a judgment P **true** for propositions. This is very straightforward, using propositions as types. A proposition is just a type in the basic theory. A proposition is true if there is some element inhabiting it, again in the basic theory. This seemingly innocent proof-irrelevance gives the subset theory its power.

The subset theory now interprets the basic judgment

$$A \textbf{ set}$$

of Martin-Löf type theory as two judgments in the underlying theory, prescribing

- a set A' in the basic theory, and
- a family of propositions $A''(x)$ **prop** $[x \in A']$, again in the basic theory.

This corresponds to our definition of specification in Section 3.

Equality of sets A, B is based on equality of the underlying sets A', B', but uses *logical* equivalence of the families A'', B''. We rejected such a choice, in favour of the decidable relation of convertibility in ECC.

The membership judgment $a \in A$ is interpreted in the obvious way: we may derive $a \in A$ if we can derive $a \in A'$ and $A''(a)$ **true** in the basic theory. This captures the essential idea, that the judgments A **set** and $a \in A$ should describe subsets of the terms in A' in the underlying theory. In this way, the proofs of the propositions $A''(a)$ are systematically suppressed. It is then relatively straightforward to see how this allows the interpretation of a subset-type constructor.

How does this compare with our approach? The resulting expressions for the various type constructors are very similar, compare for example the definition of exponential for second-order deliverables with the Π-type in the subset interpretation. We consider explicit proofs of the propositional parts of our specifications, whereas we need only know that some proof may be derived in the subset theory. However, this seems to be one of the limitations of their approach, in that we only know that certain *derivations* in the subset theory arise from certain other derivations. Our use of Σ-types, by contrast, means that we can represent the derivations of deliverables as actual *terms* within ECC, using the definable combinators which code up the explicit translation. The price we pay seems to be that we have to work with a rather clumsy language for these terms, as opposed to the conceptual elegance of reusing the basic language of types and terms in the subset theory.

The NuPrl system, developed by Constable and his co-workers [5], is based on early versions of Martin-Löf's type theory. In particular, the underlying term calculus is untyped, and the system has extensional equality types. This has the advantage of suppressing some irrelevant information in proofs. It also overcomes the limitations on the use of an explicit subset-type constructor in a theory with intensional equality, exposed in [33, 34]. The sovereign disadvantage is that the basic judgments of the theory become undecidable, coupled with a proliferation of well-formedness conditions in the application of the rules.

Recently, Hayashi has also proposed a system based on realisability, which abandons the usual type constructors Π, Σ, on which most work to date on type theory has been based, in favour of a more set-theoretic style, with union, intersection and singleton types [13]. The system he considers is, however, ingenious enough to represent dependent products and dependent sums. At the same time, the typing rules for union and intersection hide information. This allows a simple translation or extraction into a programming language with a polymorphic type discipline. Singleton types seem essential in achieving this harmony between the type system and the underlying untyped terms.

Pavlovič, in his thesis [32], elaborates in categorical terms a theory of constructions in which programs do not depend on proofs of logical propositions. As with the models of Constructions considered by Hyland and Pitts [18], the emphasis is on extensional systems, rather than the intensional system we work with here. Proof-theoretic properties seem to be regarded as something "...an implementation would have to answer" [32, p.8].

8 Conclusions

We have described a proof development method which takes as its basic entity a program plus its correctness proof. If the programs are functional these program proof-pairs admit exactly the same combination operators as do functions. We have studied the mathematical theory of such program proof-pairs and the operations which combine them, and we have considered the case where the pre- and postconditions are contextually linked to express a relationship between input and output. The combination operations have been coded within the Lego proof system; a number of examples have been developed in Lego, and some metatheoretic results about the operations have been proved in Lego.

Acknowledgements

We would like to thank Randy Pollack for the Lego system and Zhaohui Luo for his contribution to its theoretical underpinning. We also thank our colleagues in the Lego Club for friendly comments and stimulation. We thank Per Martin-Löf, Gordon Plotkin and Susumu Hayashi for helpful comments on presentations of this work.

References

[1] J.Bénabou, *Fibred categories and the foundations of naïve category theory*, JSL, 1985.

[2] S. Berardi, *Type Dependence and Constructive Mathematics*, Ph.D. thesis, Dipartimento di Informatica, Torino, Italy 1990.

[3] N.G. de Bruijn, *A survey of the project AUTOMATH*, in: [36].

[4] R.M.Burstall and J.H.McKinna, *Deliverables: an approach to program development in Constructions*, in [17], also available as a University of Edinburgh technical report ECS-LFCS-91-133.

[5] R.Constable *et al.*, *Implementing Mathematics with the NuPrl Proof Development System*, Prentice-Hall, New Jersey, 1986.

[6] T.Coquand and G.Huet, *Constructions: a Higher-order Proof system for mechanizing mathematics*, in: Proceedings *EUROCAL* '85, LNCS 203, Springer-Verlag, 1985.

[7] T.Coquand, *Metamathematical Investigations of a Calculus of Constructions*, in: [16].

[8] P-L.Curien, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman Research Notes in Theoretical Computer Science, Pitman, London, 1986.

[9] E.W.Dijkstra, *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*, in: Communications of the ACM, Vol. 18, 1975.

[10] J-Y.Girard, *Interpretation fonctionelle et élimination des coupures dans l'arithmétique de l'ordre supérieure*, thesis, University of Paris VII, 1972.

[11] R.Harper and R.A.Pollack, *Type checking with universes*, in: Theoretical Computer Science, Vol. 89, North-Holland, Amsterdam, 1991.

[12] S.Hayashi, *Adjunction of semifunctors: categorical structures in nonextensional lambda calculus*, in Theoretical Computer Science, Vol. 41, North-Holland, Amsterdam, 1985.

[13] S.Hayashi, *Singleton, Union and Intersection Types for Program Extraction*, in: Proceedings of TACS '91, Sendai, Japan, Springer LNCS 526, Springer-Verlag, 1991.

[14] C.A.R.Hoare, *An axiomatic basis for computer programming*, in: Communications of the ACM, Vol. 12, 1969.

[15] W.A.Howard, *The "formulae-as-types" notion of construction*, in: [36].

[16] G.Huet, T.Coquand, C.Paulin-Mohring *et al.*, *The Calculus of Constructions, Version 4.10, Documentation and user's manual*, Rapports Techniques no.110, Projet Formel, INRIA-Rocquencourt, Paris, August 1989.

[17] G.Huet and G.Plotkin, eds. *Electronic Proceedings of the First Annual BRA Workshop on Logical Frameworks, Antibes, May 1990,*, distributed electronically to participating BRA sites, January 1991.

[18] J.M.E.Hyland and A.M.Pitts, *The Theory of Constructions: Categorical Semantics and Topos-theoretic models*, in: Proceedings of the AMS Conference on Categories in Computer Science, Boulder, Colorado, 1986.

[19] P.T.Johnstone and R.Paré, eds., *Indexed Categories and their Applications*, Springer LNM 661, Springer-Verlag, 1978.

[20] J.Lambek and P.J.Scott, *An Introduction to Higher-Order Categorical Logic*, Cambridge Studies in Advanced Mathematics no. 7, Cambridge University Press, Cambridge, England, 1986.

[21] Z.Luo, *ECC, an Extended Calculus of Constructions*, in: Proceedings of the Fourth IEEE Conference on Logic in Computer Science, Asilomar, California, 1989.

[22] Z.Luo, *An Extended Calculus of Constructions*, Ph.D. Thesis, Department of Computer Science, University of Edinburgh, June 1990.

[23] Z.Luo, *Program Specification and Data Refinement in Type Theory*, Technical Report ECS-LFCS-90-131, Department of Computer Science, University of Edinburgh, January 1991.

[24] Z.Luo and R.Pollack, *LEGO Proof Development System: User's Manual*, LFCS Technical Report ECS–LFCS–92-211, 1992.

[25] J.H.McKinna, *Deliverables: a categorical approach to program development in type theory*, Ph.D. thesis, University of Edinburgh, 1992.

[26] P.Martin-Löf, *An Intuitionistic Theory of Types: Predicative part*, in: Logic Colloquium 73, North-Holland, Amsterdam, 1975.

[27] P.Martin-Löf, *Constructive Mathematics and Computer Programming*, in: proceedings of the Conference on Logic, Philosophy and Methodology of Science VI, 1979, North-Holland, Amsterdam, 1982.

[28] M.Mendler, *The Logic of Design*, Ph.D. thesis, University of Edinburgh, forthcoming, 1992.

[29] B.Nordström, K.Petersson, and J.Smith, *Programming in Martin-Löf's type theory*, Oxford University Press, 1990.

[30] C.Paulin-Mohring, *Extracting F_ω's programs from proofs in the Calculus of Constructions*, in: Proceedings POPL89, ACM, 1989.

[31] C.Paulin-Mohring and B.Werner, *Extracting and Executing Programs developed in the Inductive Constructions System: a Progress Report*, in: [17].

[32] D.Pavlovič, *Predicates and Fibrations*, proefschrift, University of Utrecht, 1990.

[33] A.Salvesen and J.Smith, *On the strength of the subset type in Martin-Löf's type theory*, in: Proceedings of the Third LICS Symposium, IEEE, 1988.

[34] A.Salvesen, *On Information Discharging and Retrieval in Martin-Löf's type theory*, Ph.D. thesis, Institute of Informatics, University of Oslo, 1989.

[35] D.Sannella, *Formal specification of ML programs*, LFCS technical report ECS-LFCS-86-15, Dept. of Computer Science, University of Edinburgh, 1986.

[36] J.P.Seldin and J.R.Hindley, eds., *To H.B.Curry, essays in Combinatory Logic, λ-calculus and Formalism*, Academic Press, 1980.

[37] S.Stenlund, *Combinators, λ-calculus and Proof Theory*, D.Reidel, Dordrecht, 1972.

Pattern Matching with Dependent Types

Thierry Coquand*
Chalmers University

Preliminary version, June 1992

Introduction

This note deals with notation in type theory. The definition of a function by pattern matching is by now common, and quite important in practice, in functional programming languages (see for instance [1]). We try here to introduce such definitions by pattern matching in Martin-Löf's logical framework.

1 Statement of the Problem

1.1 A Short Presentation of Martin-Löf's Logical Framework

For a more complete presentation of Martin-Löf's logical framework, which is implemented in ALF, we refer to the book "Programming in Martin-Löf's Type Theory" [16], chapter 19 and 20. We recall that each type T is of the form $(x_1 : A_1, \ldots, x_n : A_n)A$ where A is Set or of the form $El(a)$. If A is of the form $El(a)$, we say that T is an **small type**, and it is a **large type** otherwise if A is Set. If a type of a term is of the form $(x_1 : A_1, \ldots, x_n : A_n)A$, we say that n is the **arity** of this term. An **instance** of a term u of arity n is a term definitionally equal to a term of the form $u(v_1, \ldots, v_n)$.

A **context** is a list of type declaration $\Gamma = x_1 : A_1, \ldots, x_n : A_n$. As in [19], we relativize all judgements of type theory with respect to a context. An **interpretation** or **contextual mapping** between two contexts $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ and $\Delta = y_1 : B_1, \ldots, y_m : B_m$ is a simulateneous substitution $S = \{y_1 := v_1, \ldots, y_m := v_m\}$ such that

$$v_1 : B_1\ (\Gamma),\ v_2 : B_2[v_1]\ (\Gamma), \ldots, v_m : B_m[v_1, \ldots, v_{m-1}]\ (\Gamma)^1.$$

We write in this case $S : \Gamma \to \Delta$. If M is an open expression in Δ, we write by simple juxtaposition MS the result of the substitution S to M. Notice that if A is a type in Δ then AS is a type in Γ, and if $a : A\ (\Delta)$, then $aS : AS\ (\Gamma)$.

If $T_1 : \Gamma_1 \to \Gamma$ and $T_2 : \Gamma_2 \to \Gamma_1$ we write $T_2; T_1 : \Gamma_2 \to \Gamma$ the composition of T_1 and T_2.

Martin-Löf's logical framework is an open framework: the user can add new constants and new computation rules.

*coquand@cs.chalmers.se

[1]If v_i is definitionally equal to y_i, we omit $y_i := v_i$ in the writing of the interpretation; thus, $\{x := 0\}$ is a contextual mapping from $y : \mathsf{N}$ to $x : \mathsf{N}, y : \mathsf{N}$ meaning $\{x := 0, y := y\}$.

For instance, we get the Σ sets by declaring the constants [2]:

$$
\begin{array}{rcl}
\Sigma & : & (X : \mathsf{Set}, (X)\mathsf{Set})\,\mathsf{Set} \\
\mathsf{pair} & : & (X : \mathsf{Set}, Y : (X)\mathsf{Set})\,(x : X, Y(x))\,\Sigma(X, Y) \\
\mathsf{split} & : & \quad (X : \mathsf{Set}, Y : (X)\mathsf{Set})\,(Z : (\Sigma(X, Y))\mathsf{Set}) \\
& & \qquad ((x : X, y : Y(x))Z(\mathsf{pair}(X, Y, x, y))) \\
& & \qquad\quad (w : \Sigma(X, Y)) \\
& & \qquad\qquad Z(w)
\end{array}
$$

and asserting the equality (which can be read as a computation rule):

$$
\mathsf{split}(A, B, Z, z, \mathsf{pair}(A, B, a, b)) = z(a, b) : Z(\mathsf{pair}(A, B, x, y))
$$

where

$$
\begin{array}{rcl}
A & : & \mathsf{Set}, \\
B & : & (A)\,\mathsf{Set}, \\
Z & : & (\Sigma(A, B))\,\mathsf{Set} \\
a & : & A, \\
b & : & B(a) \\
z & : & (x : A, y : B(x))\,Z(\mathsf{pair}(A, B, a, b))
\end{array}
$$

The usual cartesian product is defined by

$$
A \times B = \Sigma(A, (x)B) : \mathsf{Set} \quad [A : \mathsf{Set}, B : \mathsf{Set}]
$$

The set of natural numbers is introduced by declaring the constants:

$$
\begin{array}{rcl}
\mathsf{N} & : & \mathsf{Set} \\
\mathsf{0} & : & \mathsf{N} \\
\mathsf{succ} & : & (\mathsf{N})\,\mathsf{N} \\
\mathsf{natrec} & : & \quad (C : (x : \mathsf{N})\,\mathsf{Set}, C(\mathsf{0}), (x : \mathsf{N}, y : C(x)\,C(\mathsf{succ}(x)), n : \mathsf{N})C(n)
\end{array}
$$

and the equalities (which can be read as computation rules):

$$
\begin{array}{l}
\mathsf{natrec}(C, x, z, \mathsf{0}) = x : C(\mathsf{0}) \\
\mathsf{natrec}(C, x, z, \mathsf{succ}(a)) = z(a, \mathsf{natrec}(C, x, z, a))
\end{array}
$$

where

$$
\begin{array}{rcl}
C & : & (x : \mathsf{N})\,\mathsf{Set} \\
x & : & C(\mathsf{0}) \\
z & : & (x : \mathsf{N}, y : C(x))\,C(\mathsf{succ}(x))
\end{array}
$$

The computation rules generate the definitional equality between terms.

[2]We allow ourselves to write in general A instead of $El(A)$.

Quite important is the distinction between **canonical** and **non-canonical constants**. In the examples above, Σ, pair N, $\mathsf{0}$ and succ are canonical constants, but split, natrec and $\times$ are non canonical.

If the type of a canonical constant C is of the form $(x_1 : A_1, \ldots, x_n : A_n)\mathsf{Set}$, we say that C is a **connective**. The meaning of a connective C is given by a set of canonical constants of types of the form $(y_1 : B_1, \ldots, y_m : B_m)El(C(a_1, \ldots, a_n))$, that are called **constructors** of C. (In the case of mutual inductive definitions, we can have a set of connectives that are simultaneously defined by a set of canonical constants.) By extension, we consider also that connectives are constructors of the type Set.

A canonical constant whose type is a small type is considered to be a primitive notion, that is self-justifying. In the example above, $\mathsf{0}$ and succ are considered to be primitive notions, and the canonical set N is defined by its set of constructors $\mathsf{0}$ and succ.

We say that a term is in **constructor form** iff it is definitionally equal to a term of the form $c(u_1, \ldots, u_n)$ where c is a constructor of arity n. The constructor c is then uniquely determined. We say that a term t is **directly structurally smaller** than a term u iff

- both u and v are of small types and of arity 0,
- u is of constructor form $c(a_1, \ldots, a_n)$ and t is definitionally equal to one a_j of arity 0 or one instance of one a_j of arity > 0.

Being **structurally smaller** is defined by taking the transitive closure of this relation.

We use in an essential way the "no confusion" property of constructors. This covers two properties. The first is that a definitional equality between two terms of the form $a(u_1, \ldots, u_n)$ and $b(v_1, \ldots, v_m)$ if a and b are two distinct constructors, cannot hold. The second is that, if c is a constructor of type $(x_1 : A_1, \ldots, x_n : A_n)A$, then the equality $c(u_1, \ldots, u_n) = c(v_1, \ldots, v_n) : A[u_1, \ldots, u_n]$ implies

$$u_1 = v_1 : A_1, \ldots, u_n = v_n : A_n[u_1, \ldots, u_{n-1}].$$

The non canonical constant $\times$ is **explicitely defined** in term of split.

The definitions of split and natrec are not explicit, and we refer to these constants as **implicitely defined** constants. The meaning of implicitely defined constants is given by their computation rules.

It is an important problem to give some criteria that ensure the correctness of the addition of new constants and computation rules. We try here to analyse this problem using the pattern matching notation introduced in functional languages (see for instance [1]).

1.2 Inductively defined connectives

We shall consider only connectives that are inductively defined The relation of being structurally smaller is then expected to be well-founded. We shall take this well-foundness property as a fundamental assumption on the constructors, without trying to analyse it further here. We simply mention that the constructors presented in [7, 8] satisfy this well-foundness property.

Here are two counter-examples.

The set

$$V : \mathsf{Set},$$

with one constructor

$$\Lambda : ((A : \mathsf{Set})(A)A)V.$$

The polymorphic identity $(A, x)x$ is of type $(A : \mathsf{Set})(A)A$, and hence the term $\Lambda((A, x)x)$ is of type V. This term is structurally smaller than itself. It follows that the relation of being structurally smaller is not well-founded.

Likewise, the set

$$U : \mathsf{Set},$$

with one constructor

$$c : (\mathsf{Set})U,$$

has to be rejected. The reason is however more subtle than for the previous counter-example. We notice first that, were this set accepted, so would be $T : (U)\mathsf{Set}$ by $T(c(X)) = X : \mathsf{Set}$. We could then introduce a set $W : \mathsf{Set}$ with only one constructor $\mathsf{sup} : (x : U)((T(x))W)W$ (which is inductively defined given U, T). But then $\mathsf{sup}(c(W), (x)x)$ is structurally smaller than itself.

The first example, suggested by a remark of Per Martin-Löf, shows that the well-foundness requirement on the relation of being structurally smaller is a stronger requirement than mere normalisation. Indeed the set V is defined by a second-order quantification, and it can be shown, by the usual reducibility method, that its addition to inductively defined sets preserves the normalisation property.

1.3 Some difficulties with the usual elimination schemas

It is known how to associate to any inductively defined connective an elimination constant, together with its computation rules. This is described for instance in [6]. One can check that all the examples of implicitely defined constants and computations rules described in [16] are of this form. A first criteria for ensuring the correctness of the addition of new constants and computation rules is to allow only the addition of such elimination constants. Experiments with restricting the addition of implicitely defined constants to be elimination constants have shown some drawbacks of this approach.

One first drawback is that we do not quite get the expected computational behaviour. If we define for instance $\mathsf{add} : (\mathsf{N}; \mathsf{N})\mathsf{N}$ by $\mathsf{add}(x, y) = \mathsf{natrec}(y, x, (u, v)\mathsf{succ}(v))$, then $\mathsf{add}(x, \mathsf{succ}(y))$ reduces to $\mathsf{succ}(\mathsf{natrec}(y, x, (u, v)\mathsf{succ}(v))$ and one needs to fold back this expression to get the expected $\mathsf{succ}(\mathsf{add}(x, y))$.

One second drawback is readability. For instance, we want to consider an object such as $\mathsf{half} : (\mathsf{N})\mathsf{N}$ defined by

$$\mathsf{half}(0) = 0,\ \mathsf{half}(\mathsf{succ}(0)) = 0,\ \mathsf{half}(\mathsf{succ}(\mathsf{succ}(x))) = \mathsf{succ}(\mathsf{half}(x)),$$

as given directly by these equations, rather than being given by an explicit definition which is a "coding" of this object in term of natrec.

The second drawback is in practice quite important. The pattern matching notation is essential in functional programming languages[3].

The next anomaly is the necessity to consider higher "sets" for defining naturally a function such as $\mathsf{inf} : (\mathsf{N}; \mathsf{N})\mathsf{N}$. It is quite surprising that, in order to justify the equations

$$\mathsf{inf}(0, y) = 0,\ \mathsf{inf}(\mathsf{succ}(x), 0) = 0,\ \mathsf{inf}(\mathsf{succ}(x), \mathsf{succ}(y)) = \mathsf{succ}(\mathsf{inf}(x, y)),$$

one needs to introduce the set of numerical functions.

Another problem appeared for inductively defined families. Given a connective with an arity > 1, there are several possible elimination constants depending on what arguments are considered to be parameters. For instance, there are two different elimination constants for the connective $\mathsf{Id} : (A : \mathsf{Set}, x, y : A)\mathsf{Set}$ of unique constructor $\mathsf{refl} : (A : \mathsf{Set}; x : A)\mathsf{Id}(A, x, x)$. In this case, it is yet unknown if these two elimination constants are equivalent.

The first and, in particular, second drawbacks are strong motivations for allowing the introduction of implicitely defined constants defined by computation rules that are pattern matching equations. This seems to solve in general the third anomaly. In an unexpected way, this seems to have some bearing on the fourth problem, as we will try to explain below.

2 A General Presentation of Pattern Matching

There are two independent requirements for the correctness of the introduction of one implicitely defined constant together with its computation rules. These requirements are only sufficient in ensuring that the constant does define a total function on the underlying datatype.

The first is the requirement that all definitions, that may be recursive, are well-founded.

The second is that the equations cover all possible cases of the arguments and do not introduce ambiguities in the computation. We ensure this by imposing the definitions to be exhaustive and mutually disjoint.

2.1 Well-founded Definitions

A simple condition ensures the fact that all definitions are well-founded, and seem furthermore sufficient in practice. Let n be the arity of the implicitely defined constant f to be defined. The condition is that there exists an index $i \leq n$ such that, for all equations $f(u_1, \ldots, u_n) = e$, and all recursive call $f(v_1, \ldots, v_n)$ of f in e, the constant f does not occur in $v_1, \ldots, v_n$ and the term v_i is structurally smaller than the term u_i.

It would be possible to give a less restrictive condition, by considering instead a lexicographic extension of the structural ordering. However, this restriction suffices to recover the usual elimination schemas. It is also quite simple to ensure that this condition holds.

Notice that this condition provides more general equations than the ones provided by the usual primitive recursive schema. In the usual primitive recursive schema indeed, the parameters cannot vary in recursive calls. This is not required here.

[3]The earliest, to our knowledge, mention of this notation appears in [2]. A proposal of extending functional language with an "inductive" case expression, which hence ensures termination, is presented in [3].

For instance, this will justify directly the following kind of definitions of a a function f : $(\mathsf{N},\mathsf{N})\mathsf{N}$.

$$f(0,m) = g(m), \;\; f(\mathsf{succ}(n),m) = h(n,m,f(n,k(n,m))),$$

where $g : (\mathsf{N})\mathsf{N}$, $h : (\mathsf{N},\mathsf{N},\mathsf{N})\mathsf{N}$ and $k : (\mathsf{N},\mathsf{N})\mathsf{N}$ are previously defined functions. Notice that the parameter m changes to $k(n,m)$ in the recursive call of f. This can be done only using the set of numerical function if we restrict ourselves the usual schema of primitive recursion (see [4]).

2.2 Covering

To analyse further the condition that the definitions are exhaustive and mutually disjoint, we introduce one notion reminiscent of a notion used in Per Martin-Löf's representation of choice sequences in type theory.

Let us motivate briefly what follows. We want to add a new implicitely defined constant f of type $(x_1 : A_1, \ldots, x_n : A_n)A$, together with a set of computation rules. Let Δ be the context $x_1 : A_1, \ldots, x_n : A_n$ of arguments of f. We only consider computation rules for f of the form

$$f(a_1, \ldots, a_n) = e : A[a_1, \ldots, a_n] \; (\Gamma),$$

with $a_1 : A_1, \ldots, a_n : A_n[a_1, \ldots, a_{n-1}]$. We can think of $a_1, \ldots, a_n$ as defining a contextual mapping $S : \Gamma \to \Delta$, and this suggests to introduce the notation $f(S) = e : AS \; (\Gamma)$ for such a computation rule.

With this notation, the conditions on a system of computation rules $f(S_j) = e_j : AS_j \; (\Gamma_j)$ will be expressed as conditions on a system of contextual mappings $S_j : \Gamma_j \to \Delta$. We want to express that such a system defines a "partition of the space defined by Δ."

We are going to analyse this problem in the same way that pattern matching in ordinary functional languages is reduced to a succession of case expressions over a variable (see [1]).

We say first that a system of contextual mapping $S_1 : \Gamma_1 \to \Delta, \ldots, S_m : \Gamma_m \to \Delta$ over a common context $\Delta = x_1 : A_1, \ldots, x_n : A_n$ is an **elementary covering** of Δ iff there exists an index $i \leq n$ such that

- all terms $x_i S_j : A_i S_j \; (\Gamma_j)$, for $j \leq m$, are in constructor form,
- if $S : \Gamma \to \Delta$ is a contextual mapping such that $x_i S$ is in constructor form, then there exists one and only one $j \leq m$ and $T : \Gamma \to \Gamma_j$ such that $S = T; S_j$.

This definition may look complicated but it is a possible way of specifying what is a case expression over the ith argument. In the case of a context with only non dependent types, we recover the usual notion of case expression as in [1]. In the general case however, we cannot keep the same notion of patterns of [1] (as the examples below will show, we need for instance to consider non linear patterns), and our abstract definition seems necessary.

An instance is the elementary covering defined by $x = 0$ and $x = \mathsf{succ}(y)$ $(y : \mathsf{N})$ of the context $x : \mathsf{N}$.

A second example is the empty set of contextual maps over the context

$$\Delta = p : \mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)).$$

This is an elementary covering. Indeed, the only constructor of the connective Id is refl, and a term of the form $\mathsf{refl}(A, u)$ cannot be of type $\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$. Otherwise, we would have

$$\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0)) = \mathsf{Id}(A, u, u),$$

and hence, because Id is a constructor, $0 = u : \mathsf{N}$ and $\mathsf{succ}(0) = u : \mathsf{N}$. But this implies $0 = \mathsf{succ}(0) : \mathsf{N}$, which does not hold, because 0 and succ are different constructors.

A more elaborate example is for the context

$$\Delta = x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y).$$

It can be checked that, if we define

$$\Gamma = x : \mathsf{N}, p : \mathsf{Id}(\mathsf{N}, x, x),$$

then the unique contextual mapping

$$\{y := x, q := \mathsf{refl}(\mathsf{N}, x)\} : \Gamma \to \Delta,$$

defines an elementary covering of Δ. Indeed, this follows from the fact that refl is the only constructor of Id and that if $\mathsf{refl}(\mathsf{N}, u)$ is of type $\mathsf{Id}(\mathsf{N}, v, w)$, we have

$$\mathsf{Id}(\mathsf{N}, u, u) = \mathsf{Id}(\mathsf{N}, v, w) : \mathsf{Set},$$

and hence, since Id is a constructor, we have $u = v : \mathsf{N}$ and $u = w : \mathsf{N}$.

We define now what it means for a system of contextual mapping $S_i : \Delta_i \to \Delta$ into a common context Δ to be a **covering** of Δ :

- the identity interpretation $\Delta \to \Delta$ is a covering of Δ,
- if $S_i : \Delta_i \to \Delta$, for $i \leq p$ is an elementary covering of Δ and $T_{ij} : \Delta_{ij} \to \Delta_i$, for $j \leq q_i$, is a covering of Δ_i, then $T_{ij}; S_i : \Delta_{ij} \to \Delta$ is a covering of Δ.

For instance $x = 0$, together with $x = \mathsf{succ}(0)$ and $x = \mathsf{succ}(\mathsf{succ}(y))$ $(y : N)$ define a covering of $x : N$.

An example of covering of the context $\Delta = x : \mathsf{N}, y : \mathsf{N}$ is given by

- $\{x := 0\} : (y : \mathsf{N}) \to \Delta$,
- $\{x := \mathsf{succ}(x_1), y := 0\} : (x_1 : \mathsf{N}) \to \Delta$ and
- $\{x := \mathsf{succ}(x_1), y := \mathsf{succ}(y_1)\} : (x_1 : \mathsf{N}, y_1 : \mathsf{N}) \to \Delta$.

If we take again our last example of an elementary covering, it can be checked that the unique contextual mapping

$$\{p := \mathsf{refl}(\mathsf{N}, x)\} : (x : \mathsf{N}) \to \Gamma,$$

is an elementary covering of Γ. Hence, the unique contextual mapping

$$\{y := x, p := \mathsf{refl}(\mathsf{N}, x), q := \mathsf{refl}(\mathsf{N}, x)\} : (x : \mathsf{N}) \to \Delta,$$

is a covering of $\Delta = x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y)$.

Following Per Martin-Löf's terminology, we call **neighbourhood** of a context any contextual map that is part of a covering of this context. The collection of neighbourhoods of a covering of a context can be thought of as defining a partition of the "space" defined by this context. This notion of neighbourhood corresponds to the notion of patterns used in functional programming languages: in the case of a context with only non dependent types, we recover exactly the notion of pattern matching described in [3, 1].

2.3 Sufficient Conditions For Correctness

The sufficient conditions ensuring the correctness of the addition of a new implicitely defined constant f of type $(x_1 : A_1, \ldots, x_n : A_n)A$, of argument context $\Delta = x_1 : A_1, \ldots, x_n : A_n$ and of computation rules of the form $f(S_j) = e_j : AS_j \ (\Gamma_j)$ are that:

- there is no nested occurence of f in e_j, and all recursive call of f are done on structurally smaller arguments than the lefthandside arguments (which can be ensured as described above),
- the system of contextual maps $S_j : \Gamma_j \to \Delta$ is a covering of Δ.

2.4 Some comments on this method

The method followed here can be described as follows. When justifying a rule

$$f : (x_1 : A_1, \ldots, x_n : A_n)A,$$

we analyse exhaustively the possible forms S of the arguments of f, and in each possible case S, we build a term e_S of type AS, using constructors and already defined constants.

We allow recursive calls of the constant we are defining, provided these calls are on structurally smaller arguments.

Naturally associated to this justification of an implicitely defined constant

$$f : (x_1 : A_1, \ldots, x_n : A_n)$$

is the following computation rule for f. If a given argument $(a_1, \ldots, a_n)$ is an instance of the case S, then the value of $f(a_1, \ldots, a_n)$ is the value of the corresponding instance of e_S. Otherwise, the argument list of f is not "instantiated enough", and $f(a_1, \ldots, a_n)$ cannot be head reduced.

2.5 Some Examples

The function $\mathsf{inf} : (\mathsf{N}, \mathsf{N})\mathsf{N}$ which is defined implicitely by:

$$\mathsf{inf}(0, y) = 0, \ \mathsf{inf}(\mathsf{succ}(x), 0) = 0, \ \mathsf{inf}(\mathsf{succ}(x), \mathsf{succ}(y)) = \mathsf{succ}(\mathsf{inf}(x, y)).$$

The recursive call is justified by the fact that it is structurally smaller on the first (or the second) argument.

It is standard how to reduce such a definition to the usual elimination rules over the type N, by using the set of numerical functions.

By contrast, it is not clear how to represent the following computation rule in term of the usual elimination rules[4]. We have seen that the unique contextual mapping

$$\{y := x, p := \mathsf{refl}(\mathsf{N}, x), q := \mathsf{refl}(\mathsf{N}, x)\} : (x : \mathsf{N}) \to \Delta,$$

is a covering of $\Delta = x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y)$. It follows that it is correct to add a new constant $f : (x, y : \mathsf{N}; p, q : \mathsf{Id}(\mathsf{N}, x, y))\mathsf{Id}(\mathsf{Id}(\mathsf{N}, x, y), p, q)$ together with the computation rule

$$f(x, x, \mathsf{refl}(\mathsf{N}, x), \mathsf{refl}(\mathsf{N}, x)) = \mathsf{refl}(\mathsf{Id}(\mathsf{N}, x, x), \mathsf{refl}(\mathsf{N}, x)) \ \ (x : \mathsf{N})$$

The next example still concerns the connective Id. As we said before, there are two possible elimination rules over this connective, depending on what arguments are considered to be paramaters.

The first one, with the first argument is a parameter, is

$$\begin{array}{lll} F & : & (A : \mathsf{Set}; C : (x, y : A; \mathsf{Id}(A, x, y))\mathsf{Set}; \\ & & d : (x : A)C(x, x, \mathsf{refl}(A, x)); a, b : A; c : \mathsf{Id}(A, a, b)) \\ & & C(a, b, c) \end{array}$$

of computation rule

$$F(A, C, d, a, a, \mathsf{refl}(A, a)) = d(a) : C(a, a, \mathsf{refl}(A, a)),$$

where

$$A : \mathsf{Set},\ C : (x, y : A; \mathsf{Id}(A, x, y))\mathsf{Set},\ d : (x : A)C(x, x, \mathsf{refl}(A, x)),\ a : A.$$

The second one, with the first two arguments are parameters, is

$$\begin{array}{lll} G & : & (A : \mathsf{Set}; a : A; C : (y : A; \mathsf{Id}(A, a, y))\mathsf{Set}; \\ & & d : C(a, \mathsf{refl}(A, a)); b : A; c : \mathsf{Id}(A, a, b)) \\ & & C(b, c) \end{array}$$

of computation rule

$$G(A, a, C, d, a, \mathsf{refl}(A, a)) = d : C(a, \mathsf{refl}(A, a)),$$

where

$$A : \mathsf{Set},\ a : A,\ C : (y : A; \mathsf{Id}(A, a, y))\mathsf{Set},\ d : C(a, \mathsf{refl}(A, a)).$$

It can be checked that both constants satisfy the sufficient conditions for correctness given above. Only the covering condition has to be checked, because there is no recursive call.

The last example is the well-founded set connective:

$$\mathsf{W} : (A : \mathsf{Set}, B : (A)\mathsf{Set})\mathsf{Set},$$

of unique constructor

$$\mathsf{sup} : (A : \mathsf{Set}, B : (A)\mathsf{Set}, a : A, u : (B(a))\mathsf{W}(A, B))\mathsf{W}(A, B).$$

[4]This problem has been independently suggested by Thomas Streicher.

We can introduce the implicitely defined constant

$$\begin{array}{lll} \mathsf{wrec} & : & (A : \mathsf{Set}, B : (A)\mathsf{Set}, C : (\mathsf{W}(A,B))\mathsf{Set}, \\ & & f : (a : A, u : (B(a))\mathsf{W}(A,B), (x : B(a))C(u(x)))C(\mathsf{sup}(A,B,u)), t : \mathsf{W}(A,B))C(t) \end{array}$$

with the computation rule

$$\mathsf{wrec}(A,B,C,f,\mathsf{sup}(A,B,a,u)) = f(a,u,(x)\mathsf{wrec}(A,B,C,f,u(x))),$$

where

$$A : \mathsf{Set}, B : (A)\mathsf{Set}, C : (\mathsf{W}(A,B))\mathsf{Set}, f : (a : A, u : (B(a))\mathsf{W}(A,B), (x : B(a))C(u(x))).$$

This is justified since $u(x)$ is structurally smaller than $\mathsf{sup}(A,B,a,u)$.

3 How to build coverings

3.1 Unification Problem

If Δ is a context, A a type in Δ, and u, v two terms in Δ of type A, we define a solution of the unification problem

$$u = v : A\ (\Delta)$$

to be a finite system of contextual mappings $S_j : \Gamma_j \to \Delta$ such that

- for all j, we have $uS_j = vS_j : AS_j\ (\Gamma_j)$, and
- if $S : \Gamma \to \Delta$ is a contextual mapping such that $uS = vS : AS\ (\Gamma)$, then there exists one and only one j and $T : \Gamma \to \Gamma_j$ such that $T; S_j = S$.

For a description of the unification problem with dependent types, see [18] and [9]. Since this problem contains already the similar problem for simply typed lambda-calculus, described in [13], we cannot expect to have a general algorithm to solve it. It is however possible to describe a simple algorithm[5], that has three possible outputs

- the system with no contextual mapping (this ensures that the unification problem has no solution),
- a system with exactly one contextual mapping (this ensures that the unification problem has a most general solution),
- the algorithm fails (which corresponds to a difficult unification problem).

[5]This algorithm is similar to the first-order unification algorithm, using the fundamental fact that constructors are one-to-one function.

3.2 Splitting Contexts

We give first a way to build elementary coverings, as it is implemented in ALF. We cannot ensure that this generates all possible elementary coverings, but it is not clear yet how to extend this algorithm, and whether such an extension is needed or not in practice.

Given a context

$$\Delta = x_1 : A_1, \ldots, x_n : A_n,$$

and an index $i \leq n$ such that A_i is a small type, we describe an operation called **splitting the context Δ along** i. This is an algorithm that tries to produce an elementary covering of Δ :

- if A_i is of arity > 0, or if A_i is not in constructor form, then the algorithm fails to produce any covering,
- otherwise, A_i is of the form $El(C(u_1, \ldots, u_n))$ and we can list all the constructors of the connectives C. For each such constructor c of type $(y_1 : B_1, \ldots, y_m : B_m)El(C(v_1, \ldots, v_m))$, we apply the previous unification algorithm for the equation

  $$C(u_1, \ldots, u_n) = C(v_1, \ldots, v_n) : \mathsf{Set}\ (x_1 : A_1, \ldots, x_{i-1} : A_{i-1}, y_1 : B_1, \ldots, y_m : B_m),$$

  and we collect all the solutions.

Given the fundamental "no confusion" property of constructor, this produces in case of success an elementary covering of Δ.

3.3 General Coverings

General coverings can now be built interactively. Given a context Δ, the user chooses an index i and tries to split Δ along i. If the system answers by giving an elementary covering, the user can then choose to split some of the new produced contexts, and so on, until the user stops eventually producing by composition a covering of Δ.

This interactive way of building coverings has been implemented in ALF, and seems in practice to be quite convenient for the user in ensuring that no cases have been forgotten during the definition of a function by pattern matching. This is in contrast with the usual presentation in functional languages, where one should write the possible cases, and the compiler warns the user that some cases have been forgotten.

The following is a semi-algorithm that checks whether or not a system of contextual mappings $S_j : \Gamma_j \rightarrow \Delta$ is a covering[6] (thanks to G. Huet).

First, the system with only the identity mapping is a covering. Otherwise, choose an index i such that all $x_i S_j$ are in constructor form. Then, if possible, split Δ along i. If the answer is an elementary covering $T_i : \Delta_i \rightarrow \Delta$ of Δ, this induces a partition of the original system $S_j : \Gamma_j \rightarrow \Delta$ into a system of mappings $\Gamma_j \rightarrow \Delta_i$. We then recursively check that each of these systems is a covering.

[6]If we think of a covering as a collection of disjoint "pieces" that form a partition of a space, this semi-algorithm solves a typical "puzzle" problem. We are given some "pieces" of a space (contextual mapping), and we try to see whether or not they form a partition of this space.

4 Addition of Subsets

Kent Petersson, and independently A. Salvesen, suggested the following notion of subsets which seems to fit nicely with the present notion of implicitely defined constants. We limit here ourselves to the description of a simple example.

The meaning of a connective, such as $\mathsf{N} : \mathsf{Set}$, is given by the set of its constructors

$$\mathsf{0} : \mathsf{N},\ \mathsf{succ} : (\mathsf{N})\mathsf{N}.$$

It is quite natural to allow the introduction of (direct) **subsets** of N, that we get simply by selecting a subset of this set of constructors. For instance, we can introduce the subset $\mathsf{ISZERO} : \mathsf{Set}$ with the only constructor $\mathsf{0}$, and the subset $\mathsf{POS} : \mathsf{Set}$ with the only constructor succ.

This notion of subsets fits well with the present way of defining a function by pattern matching, where one important step is to list the constructors of a given connective.

For instance, the unique computation rule defines then correctly an implicitely defined function $p : (\mathsf{POS})\mathsf{N}$.

$$p(\mathsf{succ}(x_1)) = x_1\ (x_1 : \mathsf{N}),$$

because the context $x : \mathsf{POS}$ is covered by the contextual mapping $x = \mathsf{succ}(x_1) : \mathsf{N}\ (x_1 : \mathsf{N})$.

We can dually allow the introduction of (direct) **supersets** of N, that we get by adding new constructors. Typically, the set of ordinals $\mathsf{Ord} : \mathsf{Set}$ extends the set N by the addition of one constructor

$$\mathsf{lim} : ((\mathsf{N})\mathsf{Ord})\mathsf{Ord}.$$

The following computation rules define then correctly an implicitely defined function $g : (\mathsf{Ord})\mathsf{N}$.

$$g(\mathsf{0}) = \mathsf{0},\ g(\mathsf{succ}(x)) = \mathsf{succ}(x),\ g(\mathsf{lim}(u)) = g(u(\mathsf{0})).$$

This definition is justified since $u(\mathsf{0})$ is structurally smaller than $\mathsf{lim}(u)$.

We can then define a general inclusion relation between connectives, by taking the transitive closure of the direct inclusion relation defined by the introduction of subsets and supersets. This is a decidable relation.

As the last example shows, the addition of subsets and supersets introduces some overloading facilities. These do not however compromise the decidability of the following problems:

- is the expression A a correct type in the context Γ?
- given a type A in the context Γ, is the expression a a correct term of type A in the context Γ?

as we can convince ourselves by noting that the usual algorithm for these problems apply almost without changes (using the decidability of the inclusion relation between connectives).

It is hoped that, with these new operations, one can represent rather faithfully the example presented in [15].

A more elaborate notion of subtypings appears in [11].

Conclusion

As an experiment of using pattern matching, we have done in ALF the Gilbreath Trick, presented by G. Huet last year [17], which is a non trivial inductive proof. This example shows well the gain in readibility that brings the pattern matching notation. While doing other experiments, it appeared that a quite useful extension of the system would be the introduction of case expressions for proofs, where the case is over a term that may not be in variable form. More generally, the goal seems to be to develop nice enough notations that will hopefully help the analysis of inductive arguments.

The method we follow here has some similarities with Lars Hallnäs notion of partial inductive definitions (see [12, 10]), and with the way proofs are represented in Elf [17]. What we do seems to correspond to a suggestion of [12] to use this notion as a "basis for a logical framework". These connections have to be precised.

In the present analysis of pattern matching, a crucial rôle is played by the "no confusion" property of constructors. In "Language and Philosophical Problems," [20], p. 163 - 167, S. Stenlund emphasizes from a philosophical perspective the importance of this property.

From a proof-theoretic viewpoint, our treatment can be characterized as fixing the meanings of the logical constants by the introduction rules. This possibility is discussed in [5], and constrasted to the dual possibility, which is to fix the meanings of logical constants by the elimination rules.

References

[1] Augustsson, L. "Compiling Pattern Matching." In Compiling Lazy Functional Languages Part II, Ph. D. Thesis, Chalmers, 1987.

[2] Burstall, R.M. "Proving properties of programs by structural induction." Computer Journal 12(1), p. 41 – 48, 1969.

[3] Burstall, R.M. "Inductively Defined Functions in Functional Programming Languages." Journal of Computer and System Sciences, vol. 34, p. 409 – 421, 1987.

[4] Colson, L. "About Primitive Recursive Algorithms." LNCS 372, p. 194 – 206, 1989.

[5] Dummett, M. (1991) *The Logical Basis of Metaphysics.* Duckworth ed.

[6] Dybjer, P. "Inductive Sets and Families in Martin-Löf's Type Theory" Chalmers Report 62, also p. 280-306 in Logical Frameworks, eds. G. Huet and G. Plotkin, Cambridge University Press, 1991.

[7] Dybjer, P. "An inversion principle for Martin-Löf 's type theory." Proceedings of the Workshop on Programming Logic in Bastad, May 1989, Programming Methodology Group Report 54, p. 177-190.

[8] Dybjer, P. "Universes and a General Notion of Simultaneous Inductive-Recursive Definition in Type Theory." in these proceedings.

[9] Elliott, C. M. "Higher-Order Unification with Dependent Function Types." p. 121 – 136, Proc. Rewriting Techniques and Applications

[10] Eriksson, L.H. "A Finitary Version of the Calculus of Partial Inductive Definitions." SICS research report, also to be published in LNCS, Proceedings of the Second Workshop on Extensions of Logic Programming.

[11] Freeman, T. and Pfenning, F. "Refinement Types for ML." to appear in ACM SIGPLAN 1991, Conference on Programming Language Design and Implementation.

[12] Hallnäs, L. "Partial Inductive Definitions." Theoretical Computer Science 87, 1991, p. 115 - 142.

[13] Huet, G. "A unification algorithm for typed λ-calculus." Theoretical Computer Science, p. 27 – 57, 1975.

[14] Huet, G. "The Gilbreath Trick: A Case Study in Axiomatization and Proof Development in the COQ Proof Assistant." Technical Report 1511, INRIA, September, 1991.

[15] Kahn, G. "Natural Semantics." INRIA Technical report, 601, 1987.

[16] Nordström B., Petersson K., Smith. J. M. (1990), *Programming in Martin-Löf Type Theory.* Oxford Science Publications, Clarendon Press, Oxford.

[17] Pfenning, F. "Logic Programming in the LF logical framework" in G.Huet and G. Plotkin, Logical Frameworks, Cambridge University Press.

[18] Pym, D. *Proofs, Search and Computation in General Logic.* Thesis, University of Edinburgh, November 1990.

[19] Ranta. A. (1988), "Constructing possible worlds," Mimeographed, University of Stockholm, to appear in Theoria.

[20] Stenlund, S. (1991), *Language and Philosophical Problems.* Routledge ed.

A proof of normalization for simply typed lambda calculus written in ALF

Catarina Coquand*

Preliminary version, august 1992

Abstract

We will in an semantic way define a normalization function by sructural induction for simply typed lambda calculus. First we write a function that evaluates the semantics of a term and then we define a coding that gives a term on normal form back. The work has been done in a fragment of type theory using the pattern matching of ALF[5][3].

Keywords and Phrases: Type Theory, Normalization, Proof Theory, Lambda Calculus.

1 Introduction

One aim of this work has been to show completely formally a proof of normalization of simply typed lambda calculus in such a way that it can, hopefully, be extended to type theory with dependent types. Another aim has been to gain experience in working in ALF and for example has this work suggested the pattern matching now implemented in ALF[3].

We will in this paper present a function by structural induction that computes the normal form of a term. This function is build up by two functions, one that evaluates the semantics of a term and one that from the semantics gives an eta-expanded term on normal form back. The idea that one could write an evaluation function like this was pointed out to me by Th. Coquand and P. Dybjer and Th. Coquand also pointed out that it was possible to write the coding back function.

The definitions below are all as in the implementation in ALF but the monomorphic type information has been taken away when this improves the readability. In ALF as it stands now there is no control that the functions are defined by structural induction, but it should be clear from the definitions that this is the case. It should also be clear that the definitions follows the schema presented in [4].

2 The theory

The theory used is Martin Löf's type theory with intensional identity (Id), product ($\times$ with the pairing constructor $< a, b >$), one element set (T with constructor tt) and cartesian product of

*Programming Methodology Group. Department of Computer Sciences. Chalmers University of Technology and University of Göteborg. S-412 96 Göteborg, Sweden e-mail catarina@cs.chalmers.se

dependent types (Π with constructor $\lambda.$). For a full description of this see [NPS90]. Observe that we do not define the elimination rules instead we will use the pattern matching of ALF.

We will overload the notation Π and λ in the text below, in fact we define a Π each for the number of abstractions we make $\Pi_n(A_1, \ldots A_n, B)$. We will also write $t \equiv t'$ instead of $\mathsf{Id}(A, t, t')$.

3 The syntax

The syntax of the lambda terms are standard except for that we have explicit substitution on the terms. Having explicit substitution is not essential in this paper but is probably usefull for future extensions.

3.1 Definitions of types

The types we have are base type and function type. The set of types

$$Type: \ Set^1$$

is introduced by:

$$\frac{}{o : Type} \qquad \frac{A, B : Type}{A{\rightarrow}B : Type}$$

Types will be named $A, B, \ldots$.

3.2 Definition of contexts

A context is a list of types and the set of context

$$Context: \ Set$$

is introduced by:

$$\frac{}{[] : Context} \qquad \frac{\Gamma : Context \qquad A : Type}{\Gamma.A : Context}$$

Context will be named Γ and Δ in the text below.

We also define the relation of extensions between contexts

$$\geq : (Context; Context) \ Set$$

with the constructors:

$$\frac{}{ext_0 : \Gamma \geq \Gamma} \qquad \frac{e : \Gamma \geq \Delta \qquad A : Type}{ext_1(e) : \Gamma.A \geq \Delta} \qquad \frac{e : \Gamma \geq \Delta \qquad A : Type}{ext_2(e) : \Gamma.A \geq \Delta.A}$$

[1] Please, do not be confused by this

Notice how the third constructor simplifies the definition of shift below.
Extensions will be named e below.

We will now define the transitivity function (or, if one wants, show the transitivity of $\geq$)

$$trans : (e_1 : \Gamma_1 \geq \Gamma_2; e_2 : \Gamma_2 \geq \Gamma_3)\ \Gamma_1 \geq \Gamma_3$$

The definition (or proof) is done by induction first on e_1 and in the case ext_2 by induction on e_2.

$$\begin{array}{rcl} trans(ext_0, e) & = & e \\ trans(ext_1(e_1), e_2) & = & ext_1(trans(e_1, e_2)) \\ trans(ext_2(e_1), ext_0) & = & ext_2(e) \\ trans(ext_2(e_1), ext_1(e_2)) & = & ext_1(trans(e_1, e_2)) \\ trans(ext_2(e_1), ext_2(e_2)) & = & ext_2(trans(e_1, e_2)) \end{array}$$

We have that $trans$ is associative, ie.

$$trans(e_1, trans(e_2, e_3)) \equiv trans(trans(e1, e2), e3)$$

we also have that

$$trans(e, ext_0) \equiv e$$

3.3 Definition of variables, terms and substitutions

As said above the terms are defined in a standard way except for the explicit substitution. A substitution from a context Δ to a context Γ (denoted $Subst(\Delta, \Gamma)$) is inuitively a list that associates to each variable of type A in Γ a term of type A in Δ. The substitutions we will have are the empty, updating, identity, composition, monotonicity and projection substitution. Variables are defined in a deBruijn-index style.

We will define the sets of variables, terms and substitutions

$$\begin{array}{r} Var : (Type; Context)\ Set \\ Term : (Type; Context)\ Set \\ Subst : (Context; Context)\ Set \end{array}$$

where the constructors of Var are

$$\frac{}{var_0 : Var(A, \Gamma.A)} \qquad \frac{v : Var(A, \Gamma)}{var_1(v) : Var(A, \Gamma.B)}$$

and the constructors of $Term$ are

$$\frac{t : Term(A, \Gamma) \qquad e : \Delta \geq \Gamma}{mon(t, e) : Term(A, \Delta)} \qquad \frac{}{c : Term(o, \Gamma)}$$

$$\frac{v : Var(A, \Gamma)}{var(v) : Term(A, \Gamma)} \qquad \frac{t : Term(B, \Gamma.A)}{lam(t) : Term(A \to B, \Gamma)}$$

$$\frac{t_1 : Term(A \to B, \Gamma) \qquad t_2 : Term(A, \Gamma)}{apply(t_1, t_2) : Term(B, \Gamma)} \qquad \frac{t : Term(A, \Gamma) \qquad s : Subst(\Delta, \Gamma)}{subst(t, s) : Term(A, \Delta)}$$

and the constructors of $Subst$ are

$$\frac{}{none : Subst(\Gamma, [])} \qquad \frac{s : Subst(\Delta, \Gamma) \qquad t : Term(A, \Delta)}{\{s, t\} : Subst(\Delta, \Gamma.A)}$$

$$\frac{}{s\_id : Subst(\Gamma, \Gamma)} \qquad \frac{s_1 : Subst(\Gamma_1, \Gamma_2) \qquad s_2 : Subst(\Gamma_2, \Gamma_3)}{s\_comp(s_1, s_2) : Subst(\Gamma_1, \Gamma_3)}$$

$$\frac{s : Subst(\Gamma, \Delta) \qquad e : \Gamma' \geq \Gamma}{s\_mon(s, e) : Subst(\Gamma', \Delta)} \qquad \frac{s : Subst(\Gamma, \Delta) \qquad e : \Delta \geq \Delta'}{s\_proj(s, e) : Subst(\Gamma', \Delta')}$$

Variables, terms and substitutions will benamed $v,\ t,\ s$ respectively.

3.4 Definition of normal terms.

We will now define the set of normal terms. Simultaneously we define the set of applicative terms, that is, terms on the form $apply^n(t, t_1, \ldots, t_n)$ where t is a variable or a constant and $t_1, \ldots, t_n$ are normal and the set of normal terms where a normal term of base type is an applicative term and normal term of function type is $lam(t)$ where t normal.

$$NF' : (Type; Context)\ Set$$
$$NF : (Type; Context)\ Set$$

with the constructors:

$$\frac{}{nf\_c : NF'(o, \Gamma)} \qquad \frac{v : Var(A, \Gamma)}{nf\_var(v) : NF'(A, \Gamma)}$$

$$\frac{t1 : NF'(A \to B, \Gamma) \qquad t2 : NF(A, \Gamma)}{nf\_apply(t1, t2) : NF'(B, \Gamma)}$$

$$\frac{NF'(o, \Gamma)}{nf\_o : NF(o, \Gamma)} \qquad \frac{t : NF(B, \Gamma.A)}{nf\_lam(t) : NF(A \to B, \Gamma)}$$

It is easy to see that there are a direct injection from the normal terms to the ordinary terms. (Here one would have liked a notion of subsets, but this does not exist in ALF ... yet.)

We will now define a "shift" - function on normal terms. Roughly "shift" on a variable takes $v : Var(A, \Gamma)$ to $var_1^n(v) : Var(A, \Gamma.A_1 \ldots A_n)$.

$$shift\_var : (Var(A, \Gamma); \Delta \geq \Gamma))\ Var(A, \Delta)$$
$$shift' : (NF'(A, \Gamma); \Delta \geq \Gamma)\ NF'(A, \Delta)$$
$$shift : (NF(A, \Gamma); \Delta \geq \Gamma)\ NF(A, \Delta)$$

In the definition below we will overload shift, it should be clear from the typing which is which. Notice in the last case of lam the crucial role of ext_2

$$
\begin{array}{rcl}
shift(v, ext_0) & = & v \\
shift(v, ext_1(e)) & = & var_1(shift(v, e)) \\
shift(var_0(A, \Gamma), ext_2(\Gamma, \Delta, e)) & = & var_0(A, \Delta) \\
shift(var_1(v), ext_2(e)) & = & var_1(shift(v, e)) \\
shift(nf\_c(\Gamma), e) & = & nf\_c(\Delta) \\
shift(nf\_var(v), e) & = & nf\_var(shift\_var(v, e)) \\
shift(nf\_apply(t_1, t_2), e) & = & nf\_apply(shift(t_1, e), shift(t_2, e)) \\
shift(nf\_o(t), e) & = & nf\_o(shift(t, e)) \\
shift(nf\_lam(t), e) & = & nf\_lam(shift(t, ext_2(e)))
\end{array}
$$

We have

$$shift(shift(t, e1), e2) \equiv shift(t, trans(e1, e2))$$

and

$$shift(t, ext_0) \equiv t$$

4 Semantics

The semantics of a term of base type is a term on normal form; for a term of function type it is for any extension of the context to take a value in the bigger context to a value in the bigger context. This definition is inspired by Kripke semantics seeing contexts with the relation of extension as possible worlds [2].

$$\mathcal{V} : (Type; Context)\ Set$$

$$
\begin{array}{rcl}
\mathcal{V}(o, \Delta) & = & NF(o, \Delta) \\
\mathcal{V}(A \to B, \Delta) & = & \Pi\ \Gamma \geq \Delta.\ \mathcal{V}(A, \Gamma) \Rightarrow\ \mathcal{V}(B, \Gamma)
\end{array}
$$

We will name values by v and u below.

We define an environment in which a term will be evaluated. The environment associates a value for each variable in a context.

$$\mathcal{E} : (Context; Context)\ Set$$

$$
\begin{array}{rcl}
\mathcal{E}([], \Delta) & = & \top \\
\mathcal{E}(\Gamma.A, \Delta) & = & \mathcal{E}(\Gamma, \Delta) \times\ \mathcal{V}(A, \Delta)
\end{array}
$$

Environments will be named ρ below.

We define a function that takes a value in Γ to a value in an extension of the context.

$$mon_{\mathcal{V}}\ : (A \in Type;\ \mathcal{V}(A, \Delta); \Gamma \geq \Delta)\ \mathcal{V}(A, \Gamma)$$

The function is defined by induction on the type.

$$\begin{aligned} mon_{\mathcal{V}}\ (o,t,e) &= shift(t,e) \\ mon_{\mathcal{V}}\ (A{\rightarrow}B,\lambda(f),e) &= \lambda\ e' \in \Delta' \geq \Delta, v \in\ \mathcal{V}(A,\Delta').f(trans(e',e),v) \end{aligned}$$

In the second case we obtained $\lambda(f)$ by pattern matching on the value that for the function type is a Π.

We have a corresponding function for the environment, defined by induction on the first context.

$$mon_{\mathcal{E}} : (\ \mathcal{E}(\Gamma,\Delta); \Delta' \geq \Delta)\ \mathcal{E}(\Gamma,\Delta')$$

$$\begin{aligned} mon_{\mathcal{E}}([],e,\rho) &= tt \\ mon_{\mathcal{E}}(\Gamma.A,e,<\rho,u>) &= <mon_{\mathcal{E}}(\Gamma,e,\rho),u> \end{aligned}$$

In the second case we obtained $<\rho,u>$ by pattern matching since the environment is of type $\mathcal{E}(\Gamma.A,\Delta)$ which is defined as a product.

We define the projection of a context that takes out a part of the environment.

$$proj_{\mathcal{E}} : (\ \mathcal{E}(\Gamma,\Delta); \Gamma \geq \Gamma')\ \mathcal{E}(\Gamma',\Delta)$$

$$\begin{aligned} proj_{\mathcal{E}}(ext_0,\rho) &= \rho \\ proj_{\mathcal{E}}(ext1(e),<\rho,u>) &= proj_{\mathcal{E}}(e,\rho) \\ proj_{\mathcal{E}}(ext2(e),<\rho,u>) &= <proj_{\mathcal{E}}(e,\rho),u> \end{aligned}$$

Now we are ready to define the semantics of a variable, term and substitution.

$$\begin{aligned} [\![]\!]_{Var} &: (Var(A,\Gamma);\ \mathcal{E}(\Gamma,\Delta))\ \mathcal{V}(A,\Delta) \\ [\![]\!]_{Term} &: (Term(A,\Gamma);\ \mathcal{E}(\Gamma,\Delta))\ \mathcal{V}(A,\Delta) \\ [\![]\!]_{Subst} &: (Subst(\Gamma,\Gamma');\ \mathcal{E}(\Gamma,\Delta))\ \mathcal{E}(\Gamma',\Delta) \end{aligned}$$

We will only use $[\![]\!]$ below, it should be clear from the typing which is which.

The semantics of a variable is its value in the environment. The semantics of a substitution is a new environment, where we intuitivaly have that if $\{t_1,\ldots t_n\}$ is a substitution from Γ to $[A_1\ldots A_n]$ and ρ an environment of type $\mathcal{E}(\Gamma,\Delta)$ then we get the new environment $<[\![t_1]\!]\rho,\ldots,[\![t_n]\!]\rho>$. The semantics of a term should be clear from the definition below.

In the definitions below ρ has the type $\mathcal{E}(\Gamma,\Delta)$.

$$\begin{aligned} [\![var_0]\!]<\rho,u> &= u \\ [\![var_1(v)]\!]<\rho,u> &= [\![v]\!]\rho \\ [\![mon(t,e)]\!]\rho &= [\![t]\!]proj_{\mathcal{E}}(e,\rho) \\ [\![c(\Gamma)]\!]\rho &= nf\_o(nf\_c(\Delta)) \\ [\![var(v)]\!]\rho &= [\![v]\!]\rho \\ [\![lam(t)]\!]\rho &= \lambda\ e \in \Delta' \geq \Delta, v \in\ \mathcal{V}(A,\Delta').[\![t]\!]<mon_{\mathcal{E}}(e,\rho),v> \\ [\![apply(t_1,t_2)]\!]\rho &= ([\![t_1]\!]\rho\ \ ext_0\ \ [\![t_2]\!]\rho)) \\ [\![subst(t,s)]\!]\rho &= [\![t]\!][\![s]\!]\rho \end{aligned}$$

$$
\begin{aligned}
[\![none]\!]\rho &= tt \\
[\![\{s,t\}]\!]\rho &= <[\![s]\!]\rho, [\![t]\!]\rho> \\
[\![s\_id]\!]\rho &= \rho \\
[\![s\_comp(s_1,s_2)]\!]\rho &= [\![s2]\!][\![s1]\!]\rho \\
[\![s\_mon(s,e)]\!]\rho &= [\![s]\!]proj_{\mathcal{E}}(e,\rho) \\
[\![s\_proj(s,e)]\!]\rho &= proj_{\mathcal{E}}(e,[\![s]\!]\rho)
\end{aligned}
$$

We will now define the coding back function *get_term* that takes a value and returns a term on normal form. We will simultaneously define a function *get_val* that takes an applicative term to a value. Intuitively *get_term* gives back a term that is eta-expanded on a syntactical level and *get_val* gives back a value that is eta-expanded on a meta level.

$$
\begin{aligned}
get\_term &: (A \in Type; \Gamma \in Context;\ \mathcal{V}(A,\Gamma))\ NF(A,\Gamma) \\
get\_val &: (A \in Type; \Gamma \in Context; NF'(A,\Gamma))\ \ \mathcal{V}(A,\Gamma)
\end{aligned}
$$

Both are defined by induction on the type.

$$
\begin{aligned}
get\_term_o(\Gamma,u) &= u \\
get\_val_o(\Gamma,t) &= t \\
get\_term_{A\to B}(\Gamma,\lambda(f)) &= nf\_lam(get\_term_B(\Gamma.A, f(ext_1(ext_0(\Gamma)), \\
&\qquad get\_val_A(\Gamma.A, nf\_var(var_0))))) \\
get\_val_{A\to B}(\Gamma,t) &= \lambda\ e \in \Delta \geq \Gamma, v \in\ \mathcal{V}(A,\Delta).\ get\_val_B(\Delta, \\
&\qquad nf\_apply(shift(t,e), get\_term_A(\Delta,v)))
\end{aligned}
$$

We are now ready to define the function that computes a term to its normal form. For this we define the identity environment

$$id_{\mathcal{E}} : (\Gamma \in Context)\ \ \mathcal{E}(\Gamma,\Gamma)$$

that for each variable in the context give its value which we compute with the *get_val* function above.

$$
\begin{aligned}
id_{\mathcal{E}}([]) &= tt \\
id_{\mathcal{E}}(\Gamma.A) &= < mon_{\mathcal{E}}(ext_1(ext_0(\Gamma)), id_{\mathcal{E}}(\Gamma)), \\
&\qquad get\_val_A(\Gamma.A, nf\_var(var_0)) >
\end{aligned}
$$

To compute the normal form

$$nf : (Term(A,\Gamma))\ NF(A,\Gamma)$$

is now only to compute the semantics of the term in the identity environment and then code the result back.

$$nf_A(\Gamma,t) \ = \ get\_term_A(\Gamma, [\![t]\!]id_{\mathcal{E}}(\Gamma))$$

5 Conclusions

We have defined a normalization function by structural induction using the pattern matching of ALF. I feel that using the pattern matching has been essential for the practical possibility to do this kind of proofs. What I would have liked to have is a notion of subsets and pattern matching inside an expression, ie some kind of case statement. Ovarloading of names would have been nice but not essential.

5.1 Future work

We will define a conversion on terms and show that a term converts to the injection of its normal form. We also want to show that if two terms converts then their semantical values are extensionally equal and that their normal forms are identical thereby getting a decision algorithm for conversion (just check that their normal forms are identical).

It might also be intresting to change this proof to use named variables.

6 Acknowledgements

I want to thank Thierry Coquand for giving me the idea of this work and for patiently answering all my questions.

References

[1] U. Berger and H. Schwichtenberg *An inverse of the evaluation functional for typed λ-calculus.* Proceedings of LICS 91

[2] Th. Coquand and J. Gallier, *A proof of strong normalization for the theory of constructions using a Kripke-like interpretation.* Proceedings of the first workshop in Logical Frameworks.

[3] Th. Coquand *Pattern Matching with Dependent Types.* In this proceedings.

[4] P. Dybjer *An inversion principle for Martin-Löf 's type theory.* Proceedings of the Workshop on Programming Logic in Bastad, May 1989, Programming Methodology Group Report 54, p. 177-190.

[5] L. Magnusson *The new implementation of ALF.* In this proceedings.

[6] B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

Virtual reduction

Vincent Danos, Laurent Regnier, Paris7

Abstract

We present a reduction of graphs whose edges are labelled by coefficients in the dynamical algebra Lambda *, the so-called virtual reduction. We show that the virtual reduction enjoys the Church-Rosser property. We give a semantic of the virtual reduction by applying the execution formula of Girard.

We then apply the preceding results to the case of pure-nets and introduce the notion of virtual cut. We end by relating the virtual cut elimination procedure with the family reduction of lambda-terms in the sense of Levy.

Natural Semantics in Coq. First experiments. - DRAFT -

Joëlle Despeyroux
INRIA, Sophia-Antipolis

André Hirschowitz
University of Nice

August 7, 1992

Abstract

In this paper, we discuss a possible implementation of a Natural Semantics of Mini-Ml into the Coq system. The aim of this experiment was to study to which point the Coq theorem prover is well suited to perform proofs in Natural Semantics. To represent our semantics, we have used the LF encoding of logics, which do not always meets the Coq requirements for definitions of inductive types. We propose here a possible solution to that problem, which will be developped in the final version of the paper.

1 Introduction

In a generic programming environment such as Centaur [BCD+88], there is a need for a powerful theorem prover to be used for describing and for reasoning on object languages and programs. In the current version of Centaur, the available theorem prover, Theo [Des92], is first-order and not powerful enough for many standard tasks. As benchmarks, let us ask for mechanical proofs of both the subject reduction theorem for Mini-Ml and the translation from Mini-Ml to cam. A natural tool to consider for such tasks is the programming language Elf [Pfe89], based on the logical framework LF [AHM87]. Indeed, the implementation of mini- ml in Elf has been carefully studied in [MP91], where a corresponding proof of the subject reduction theorem has been given. However, this proof relies on an induction principle which is defined at the meta-level. A natural tool to consider then is the theorem prover Coq [DFH+91], based on the Calculus of Constructions (CoC) enriched with inductively defined types. Indeed, on one hand Coq fully encompasses LF, and on the second hand, it offers a powerful notion of inductive type, hence it should be able to provide for instance a fully mechanical proof of the subject reduction theorem for Mini-Ml. In this context, we raise three questions:

1. Is coq powerful enough to perform for instance the benchmarks mentioned above?
2. What could be an efficient programming style for the implementation of natural semantics in coq?
3. Which tools or facilities should be coined and added to coq (or build over it) in order to make it more suitable for semantics of programming languages, more precisely to make it available in Centaur as a privileged, efficient and user-friendly theorem prover to be used for describing and for reasoning on languages and programs?

In this paper, we report on our first attempt to derive under coq a fully mechanical proof of the subject reduction theorem for Mini-Ml.

This attempt is based on an implementation of Mini-Ml directly translated from the one given in LF in [MP91]. The main feature of this implementation is that the chosen syntax is higher- order.

Let us describe the outcome of this first attempt in terms of the questions listed above:

1. We have produced under coq a fairly simple proof of the subject reduction theorem for Mini-Ml, relying upon two sets of axioms. The first one expresses that equality for terms of the object language is of a syntactic nature. The second one expresses the "only if" part of our semantic definition, in other words it inverts our definition. We were not able to prove these axioms, because the chosen higher-order syntax cannot be defined as an inductive type, which deprives us the suitable induction principle and the corresponding match constructor. Hence coq will be powerful enough for the above mentioned goal when we will solve the following puzzle: inside coq, we shall provide our higher- order syntax with some kind of induction principle (over the structure of the terms) and with a match constructor and this raises three problems: -what is the "right" induction principle in this case is not evident. - any match constructor will enlarge the syntax with new terms which should be excluded through an appropriate selection process. - once designed, the induction principle, the match constructor and the selection process have to be legitimated and packed in a user-friendly way. We propose below the basis of a unified solution for this puzzle.

2. As for programming style, the main choice we have considered concerns syntax. There are clearly two different approaches: -higher-order syntax, as we have experienced. Here, the higher-order features of the object language are directly implemented as such in the meta-language; hence, and this is the main advantage of this approach, the semantic rules are very simple. The main drawback for this approach is that some of the various objects we wish to define (for instance exp or type_of) are not inductively defined in coq's sense. While this doesn't affect the simplicity of semantic rules, it affects seriously, as explained above, the way proofs concerning the object language can be built. Our current conclusion here is this: for a user- friendly study of languages described by a higher-order syntax, coq at least need both an extension of the induction principle to some class, to be introduced, of "weakly-inductive" definitions and the automatic (lazy?) generation of some standard consequences of this stronger induction principle.

   -first-order syntax: this is the opposite choice. Its main advantage is that it allows the object language to be introduced as an inductive type, so that we can express the semantics through recursive definitions and build our proofs uniformly by recursion on the subject. Here, the problem is no more with proofs but with the description of the semantics. The natural solution seems to be through a suitable coercion of first-order terms, once introduced as an inductive type, to higher-order. Our conclusion here is the need for such a generic coercion. We do not explore further this approach since the next seems simpler in any respect.

   We propose a new, intermediate approach, which we call *middle-order syntax*. In first-order syntax, object lambda-expressions are of type say (exp $\rightarrow$ exp $\rightarrow$ exp), while in higher-order syntax, they are of type ((exp $\rightarrow$ exp) $\rightarrow$ exp), and this rules the type exp out of inductive types. Our intermediate solution here is to type object lambda-expressions with ((ident $\rightarrow$ exp) $\rightarrow$ exp). In this approach, the type exp is still inductive, hence it shares the advantage of the first-order syntax with respect to proofs. As for the description of the semantics, it still needs a (generic) coercion to higher- order but this coercion is much easier since bindings are already understood by middle-order syntax. Indeed, here we only need to transform terms of type (ident $\rightarrow$ exp) into the corresponding terms of type (exp $\rightarrow$ exp). This is the basis of our solution for the above mentioned puzzle.

3. The solution sketched above is far from reaching the current standards of user-friendliness under Centaur. Hence it seems necessary to build a top-level over coq, designed for the above mentioned tasks. We now list some of the features which would be welcome for this top-level.

   - at first we mention man-machine interface issues: the top-level should integrate current progress performed in our team on man-machine interface for theorem provers: proofs pretty-printed in english, and search directed from this pretty-printed proof through the mouse.
   - next the top-level should accept higher-order syntax and generate the corresponding middle-order syntax together with the associated induction principle, match operator and selection process. The middle-order syntax and the selection process should be hidden to the user.
   - moreover, for each syntactic definition, a 'unicity package' of rules should be generated expressing that equality for terms of the defined type is of a syntactic nature.
   - finally, for each definition, an 'inverse package' of rules should be generated expressing the "only if" part of the definition. This program raises corresponding theoretical issues,namely:
   - generic translation of higher-order to middle-order.

The rest of the paper is organized as follows: Section 2 is devoted to our proof of our subject reduction theorem. Section 3 is devoted to middle-order syntax. In section 4, we develop the discussion started above on the planned top-level.

2 A first experiment

2.1 The Mini-Ml language

Mini-ML [CDDK86] consists in the purely applicative part of ML, more precisely a simply typed λ-calculus with polymorphism, constants, products, conditionals, and recursive function definitions. We consider here the slight variation of Mini-Ml defined in [MP91], where patterns are replaced with explicit projections.

Simple programs in Mini-ML are for example, in concrete syntax:

$$\begin{array}{l}\mathsf{let}\ u = \lambda x.x \\ \mathsf{in}\ (u\ u)\end{array}$$

or the following simultaneous recursive definition:

$$\begin{array}{ll}\mathsf{letrec}\ (\mathit{even}, \mathit{odd}) = & (\lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ \mathit{true}\ \mathsf{else}\ \mathit{odd}\ (x-1), \\ & \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ \mathit{false}\ \mathsf{else}\ \mathit{even}\ (x-1)) \\ \mathsf{in}\ \mathit{even}\ 3 & \end{array}$$

The abstract syntax of Mini-Ml is as follows (identifiers and naturals are predefined sorts):

sorts		exp, ident		
subsorts		exp ⊃ ident		
constructors				
id	:	identifiers	$\rightarrow$	ident
$number$	:	naturals	$\rightarrow$	exp
$true, false$	:		$\rightarrow$	exp
$null, fst, snd$	:	exp	$\rightarrow$	exp
$apply, mlpair$	:	exp $\times$ exp	$\rightarrow$	exp
$lambda$	:	ident $\times$ exp	$\rightarrow$	exp
$let, letrec$	:	ident $\times$ exp $\times$ exp	$\rightarrow$	exp
if	:	exp $\times$ exp $\times$ exp	$\rightarrow$	exp

Numerous descriptions of the semantics -both static and dynamic semantics- of Mini-Ml can be found in the litterature. The type inference system of Mini-Ml has been first described by Damas and Milner [DM82]. In [Har90], R. Harper consider several presentations of this type system, together with their encoding in LF. An implementation of both static and dynamic semantics of Mini-Ml has been given both in the first-order language Typol, in [CDDK86], and in the higher-order language Elf, in [MP91].

2.2 The Mini-Ml implementation in Elf

Since our Coq implementation is based on the Elf implementation, we very briefly review this Elf implementation [MP91] in this subsection. The Coq implementation is described in the next subsection. Both implementations will be given in annex in the final version of the paper.

2.2.1 Higher-order abstract syntax

The Elf implementation of Mini-Ml describes the Mini-Ml language, using higher-order abstract syntax. The idea of this approach is to use meta-level variables to represent the object-level variables. The meta-level λ-binder is then naturally used to represent the object-level binding operators. This could seem non-natural at first sight, but has decisive advantages. The user can use the meta-level application to implement substitutions. The user can avoid the problem of α-conversion, that the meta-language has solved for him, once for all.

The syntax of Mini-ML is implemented in Elf by the following higher-order abstract syntax:

$$\begin{array}{lcl}
exp & : & type. \\
true & : & exp. \\
false & : & exp. \\
zero & : & exp. \\
succ & : & exp \to exp. \\
If & : & exp \to exp \to exp \to exp. \\
null & : & exp. \\
lam & : & (exp \to exp) \to exp. \\
app & : & exp \to exp \to exp. \\
mlpair & : & exp \to exp \to exp. \\
fst & : & exp \to exp. \\
snd & : & exp \to exp. \\
let & : & exp \to (exp \to exp) \to exp. \\
letrec & : & (exp \to exp) \to (exp \to exp) \to exp. \\
fix & : & (exp \to exp) \to exp.
\end{array}$$

In the case of Mini-Ml, we can mechanically translate a program from the Mini-Ml syntax to the Elf syntax. The two examples given above will become, in the Elf implementation (a λ-expression is denoted in Elf as [_]_):

$$let\ (lam([x]x), [u](app\ u\ u))$$

and:

$$\begin{array}{l}
letrec \\
\quad ([even_odd]\ (lam\ ([x]\ if\ (x = 0, true, (app\ (app\ (snd\ even_odd))\ (x-1)))), \\
\qquad\qquad\qquad lam\ ([x]\ if\ (x = 0, false, (app\ (app\ (fst\ even_odd))\ (x-1))))), \\
\quad [even_odd]
\end{array}$$

2.2.2 The type inference system in Elf

[...]

2.2.3 The evaluation rules in Elf

[...]

2.3 A Mini-Ml implementation in Coq

It is now well known (Henk Barendregt. GTS) that the Edinburgh Logical Framework can be viewed as a sub-system of the Calculus of Constructions. Since we shall use it in this subsection, let us call $(\_)^*$ the injection from Elf to Coq, that is induced by this view. The implementation we consider here is this natural translation from the Elf implementation, into Coq, except, maybe, for boolean and natural expressions. All the propositions written in this subsection should be more formally stated, and proved, in the final version of the paper.

2.3.1 Syntax

Let us first translate the Elf syntax into Coq, without 'changing' it. In the definition of the constructor $lam : (exp \to exp) \to exp$, the first occurrence of exp is negative. Such kind of type is not allowed in Coq as an inductive type. We must 'define' the type exp by a list of 'Parameters'. We obtain the following syntax, from which we only give here the first lines:

$$
\begin{array}{lcl}
\textsf{Parameter } exp & : & Set. \\
\textsf{Parameter } true & : & exp. \\
\textsf{Parameter } false & : & exp. \\
\textsf{Parameter } zero & : & exp. \\
\textsf{Parameter } succ & : & exp \to exp. \\
\textsf{Parameter } lam & : & (exp \to exp) \to exp. \\
\textsf{Parameter } app & : & exp \to exp \to exp. \\
\ldots & &
\end{array}
$$

The following informal proposition states that this Coq implementation of the Mini-Ml syntax is adequate with respect to the corresponding Elf implementation given above.

Proposition 1. For all closed normal term t in Elf such that $\vdash_{Elf} t : exp$, we have $\vdash_{Coq} t^* : exp^*$. For all closed normal term t' in Coq such that $\vdash_{Coq} t' : exp^*$, there exists a term t in Elf,
with $t' = t^*$, such that $\vdash_{Elf} t : exp$.

Now, it seems more convenient to use the predefined Coq inductive set *bool* and *nat* to represent the Mini-Ml boolean and natural expressions, rather than redefining them. Here is the implementation that we have chosen:

$$
\begin{array}{lcl}
\textsf{Parameter } exp & : & Set. \\
\textsf{Parameter } mlbool & : & bool \to exp. \\
\textsf{Parameter } mlnat & : & nat \to exp. \\
\textsf{Parameter } lam & : & (exp \to exp) \to exp. \\
\textsf{Parameter } app & : & exp \to exp \to exp. \\
\ldots & &
\end{array}
$$

The variation introduced is not a minor variation. vThe term $(lam\ (succ\ zero))$, for example, is an expression accepted by the first syntax, and refused by the second syntax. On the contrary, induction on booleans, and/or on naturals, is allowed by the second syntax, and not allowed by the first syntax. Another adequacy proposition, of the new syntax, with respect to the original language, is necessary.

2.3.2 The type inference system

Mini-Ml types are represented as first order terms, which are exactly the same terms as in the Elf implementation:

$$
\begin{array}{ll}
\mathsf{Inductive} & \mathsf{Set}\ mltype \\
& = tbool : mltype \\
& \mid tnat : mltype \\
& \mid arrow : mltype \to mltype \to mltype \\
& \mid cross : mltype \to mltype \to mltype.
\end{array}
$$

Our type-inference rules of Mini-Ml only differ from the Elf version in the treatment of the booleans and naturals. The rules are as follows:

$$
\begin{array}{ll}
\mathsf{Parameter}\ type: & exp \to mltype \to \mathsf{Prop}. \\
\mathsf{Parameter}\ type\_bool: & (e : bool)(type(mlbool\ e)\ tbool). \\
\mathsf{Parameter}\ type\_nat: & (e : nat)(type(mlnat\ e)\ tnat). \\
\mathsf{Parameter}\ type\_If: & (e, e_1, e_2 : exp)(t : mltype) \\
& (type\ e\ tbool) \to (type\ e_1\ t) \to (type\ e_2\ t) \to (type\ (If\ e\ e_1\ e_2)\ t). \\
\mathsf{Parameter}\ type\_null: & (type\ null\ (arrow\ tnat\ tbool)). \\
\mathsf{Parameter}\ type\_lam: & (E : exp \to exp)(t_1, t_2 : mltype) \\
& ((x : exp)(type\ x\ t_1) \to (type\ (E\ x)\ t_2)) \\
& \to (type\ (lam\ E)\ (arrow\ t_1\ t_2)). \\
\mathsf{Parameter}\ type\_app: & (e_1, e_2 : exp)(t_1, t_2 : mltype) \\
& (type\ e_1\ (arrow\ t_2\ t_1)) \to (type\ e_2\ t_2) \to (type\ (app\ e_1\ e_2)\ t_1). \\
\mathsf{Parameter}\ type\_mlpair: & (e_1, e_2 : exp)(t_1, t_2 : mltype) \\
& (type\ e_1\ t_1) \to (type\ e_2\ t_2) \to (type\ (mlpair\ e_1\ e_2)\ (cross\ t_1\ t_2)). \\
\mathsf{Parameter}\ type\_fst: & (e : exp)(t_1, t_2 : mltype) \\
& (type\ e\ (cross\ t_1\ t_2)) \to (type\ (fst\ e)\ t_1). \\
\mathsf{Parameter}\ type\_snd: & (e : exp)(t_1, t_2 : mltype) \\
& (type\ e\ (cross\ t_1\ t_2)) \to (type\ (snd\ e)\ t_2). \\
\mathsf{Parameter}\ type\_let: & (e_1 : exp)(E : exp \to exp)(t0, t : mltype) \\
& (type\ e_1\ t0) \\
& \to ((x : exp)((t_1 : mltype)(type\ e_1\ t_1) \\
& \to (type\ x\ t_1)) \to (type\ (E\ x)\ t)) \\
& \to (type\ (let\ e_1\ E)\ t). \\
\mathsf{Parameter}\ type\_letrec: & (E_1, E : exp \to exp)(t0, t : mltype) \\
& (type\ (fix\ E_1)\ t0) \\
& \to ((x : exp)((t_1 : mltype)(type\ (fix\ E_1)\ t_1) \\
& \to (type\ x\ t_1)) \to (type\ (E\ x)\ t)) \\
& \to (type\ (letrec\ E_1\ E)\ t). \\
\mathsf{Parameter}\ type\_fix: & (E : exp \to exp)(t : mltype) \\
& ((x : exp)(type\ x\ t) \to (type\ (E\ x)\ t)) \to (type\ (fix\ \ E)\ t).
\end{array}
$$

Rules type_lam, type_let, type_letrec and type_fix both contain an occurrence of *type* in a negative position that forbids us to define the judgement *type* as an inductive type.

Proposition 2. For all closed normal terms e and t in Elf such that $\vdash_{Elf} e : exp$ and $\vdash_{Elf} t : mltype$, we have $\vdash_{Elf} (type\ e\ t) \ \Rightarrow \vdash_{Coq} (type^*\ e^*\ t^*)$.
For all closed normal terms e' and t' in Coq such that $\vdash_{Coq} e' : exp^*$, $\vdash_{Coq} t' : mltype^*$, and $\vdash_{Coq} (type^*\ e'\ t')$, there exist two terms e and t in Elf such that $e' = e^*$, $t' = t^*$, and $\vdash_{Elf} (type\ e\ t)$.

2.3.3 The evaluation rules in Coq

Here again, our definitions of both the semantic values of Mini-Ml, and the evaluation rules for Mini-Ml, only differs from the Elf definition, on the booleans and naturals.

$$
\begin{array}{ll}
\textsf{Inductive} & \textsf{Definition } value : exp \to \textsf{Prop} \\
& = val\_bool : (e : bool)(value\ (mlbool\ e)) \\
& \mid val\_nat : (e : nat)(value\ (mlnat\ e)) \\
& \mid val\_null : (value\ null) \\
& \mid val\_lam : (E : exp \to exp)(value\ (lam\ E)) \\
& \mid val\_mlpair : (E_1, E_2 : exp)(value\ E_1) \to (value\ E_2) \to (value\ (mlpair\ E_1\ E_2)).
\end{array}
$$

The evaluation rules for Mini-Ml are as follows:

$$
\begin{array}{ll}
\textsf{Inductive} & \textsf{Definition } eval : exp \to exp \to \textsf{Prop} \\
& = eval\_bool : (e : bool)(eval\ (mlbool\ e)(mlbool\ e)) \\
& \mid eval\_nat : (e : nat)(eval\ (mlnat\ e)\ (mlnat\ e)) \\
& \mid eval\_If\_t : (e, e_1, e_2, v_1 : exp) \\
& \quad (eval\ e\ (mlbool\ true)) \to (eval\ e_1\ v_1) \to (eval\ (If\ e\ e_1\ e_2)\ v_1) \\
& \mid eval\_If\_f : (e, e_1, e_2, v_2 : exp) \\
& \quad (eval\ e\ (mlbool\ false)) \to (eval\ e_2\ v_2) \to (eval\ (If\ e\ e_1\ e_2)\ v_2) \\
& \mid eval\_null : (eval\ null\ null) \\
& \mid eval\_lam : (E : exp \to exp)(eval\ (lam\ E)\ (lam\ E)) \\
& \mid eval\_app : (E : exp \to exp)(e_1, e_2, v, v_2 : exp) \\
& \quad (eval\ e_1\ (lam\ E)) \to (eval\ e_2\ v_2) \to (eval\ (E\ v_2)\ v) \to (eval\ (app\ e_1\ e_2)\ v) \\
& \mid eval\_app\_null\_t : (e_1, e_2 : exp) \\
& \quad (eval\ e_1\ null) \to (eval\ e_2\ (mlnat\ O)) \to (eval\ (app\ e_1\ e_2)\ (mlbool\ true)) \\
& \mid eval\_app\_null\_f : (e_1, e_2 : exp)(n : nat) \\
& \quad (eval\ e_1\ null) \to (eval\ e_2\ (mlnat\ (S\ n))) \to (eval\ (app\ e_1\ e_2)(mlbool\ false)) \\
& \mid eval\_mlpair : (e_1, e_2, v_1, v_2 : exp) \\
& \quad (eval\ e_1\ v_1) \to (eval\ e_2\ v_2) \to (eval\ (mlpair\ e_1\ e_2)\ (mlpair\ v_1\ v_2)) \\
& \mid eval\_fst : (e, v_1, v_2 : exp) \\
& \quad (eval\ e\ (mlpair\ v_1\ v_2)) \to (eval\ (fst\ e)\ v_1) \\
& \mid eval\_snd : (e, v_1, v_2 : exp) \\
& \quad (eval\ e\ (mlpair\ v_1\ v_2)) \to (eval\ (snd\ e)\ v_2) \\
& \mid eval\_let : (e_1, v_1, v : exp)(E : exp \to exp) \\
& \quad (eval\ e_1\ v_1) \to (eval\ (E\ v_1)\ v) \to (eval\ (let\ e_1\ E)\ v) \\
& \mid eval\_letrec : (v_1, v : exp)(E_1, E : exp \to exp) \\
& \quad (eval\ (fix\ E_1)\ v_1) \to (eval\ (E\ v_1)\ v) \to (eval\ (letrec\ E_1\ E)\ v) \\
& \mid eval\_fix : (v : exp)(E : exp \to exp) \\
& \quad (eval\ (E\ (fix\ E))\ v) \to (eval\ (fix\ E)\ v).
\end{array}
$$

Proposition 3. For all closed normal terms e and v in Elf such that $\vdash_{Elf} e : exp$ and $\vdash_{Elf} v : exp$, we have $\vdash_{Elf} (eval\ e\ v) \ \Rightarrow \vdash_{Coq} (eval^*\ e^*\ v^*)$.
For all closed normal terms e' and v' in Coq such that $\vdash_{Coq} e' : exp^*$, $\vdash_{Coq} v' : exp^*$, and $\vdash_{Coq} (eval\ e'\ v')$, there exist two terms e and v in Elf such that $e' = e^*$, $v' = v^*$, and $\vdash_{Elf} (eval^*\ e\ v)$.

2.4 Proof of the subject reduction theorem

The Subject Reduction Theorem for our Mini-Ml example can be stated as follows:

$$\textsf{Theorem:}\ (e, v : exp)(eval\ e\ v) \to (t : mltype)(type\ e\ t) \to (type\ v\ t)$$

The complete proof of the theorem is given in annexB. It proceeds by induction on the definition of eval. It uses both the unicity axioms on *exp* presented later on in the section 'top-level', and the inversion of the *type* definition presented as axioms in annexB.

2.4.1 Comparaison with the Elf proof

[...]

3 Middle-order implementation of mini-ml

3.1 Syntax

Our syntax uses the empty type for bound variables.

$$Inductive\ Set\ empty = .$$

This type has no value, but it has an identity $[x : empty]x$. More generally, it has a natural morphism towards any Set:

$$\begin{array}{ll} \textsf{Definition} & init : (T : Set)(empty \to T) \\ & = [T : Set][x : empty] < T > Match\ x\ with. \end{array}$$

Further examples of functions related to the empty type are:

$$\textsf{Definition}\ pr_{2.1} = [x1 : empty][x2 : empty]x1$$

$$\textsf{Definition}\ pr_{2.2} = [x1 : empty][x2 : empty]x2$$

Our middle-order syntax for the Mini-Ml language is as follows:

$$\begin{array}{ll} \textsf{Inductive} & \textsf{Set}\ mexp \\ & = mlbool : bool \to mexp \\ & |mlnat : nat \to mexp \\ & |lam : (empty \to mexp) \to mexp \\ & |app : mexp \to mexp \to mexp \\ & |let : mexp \to (empty \to mexp) \to mexp \\ & |\cdots \end{array}$$

For instance the term $let\ u = \lambda x.x\ in\ (u\ u)$ is encoded as $(let\ ((init\ mexp), [x : empty](app\ x\ x)))$. In order to express that our syntax is correct, we need the type, denoted $(list\ t\ n)$, of lists of length n of terms of a given type t, together with the corresponding cons, car and cdr operators.

Now our syntax is correct in the following sense: there is a unique natural collection of bijections $(bij\ n)$ between $(list\ exp\ n) \to exp$ and $(list\ empty\ n) \to mexp$. This proposition would be false for a *lam* operator of type:

$$lam : (id \to mexp) \to mexp$$

This collection of bijections is 'natural' in the sense that the following properties hold (we only give here only one of these properties):

$$\begin{aligned}&(f:(list\ exp\ (S\ n)) \to exp)((bij\ n, [x:(list\ exp\ n)](lam\ [y:exp]\ (f\ (cons\ y\ x))))\\ &\quad = [x:(list\ empty\ n)](lam\ [y:empty](bij\ (S\ n)\ (f\ (cons\ y\ x))))\end{aligned}$$

The bijection $(bij\ 0)$ is the desired bijection between exp and mexp. This is the first instance where in order to understand higher-order syntax, we enlarge our scope to terms depending on a list of variables. This feature is recurrent in our work.

3.2 Coercion

Thus, the middle-order syntax is adequate but not suited for semantics. In order to express the semantics for the term, for instance, $(app\ (lam\ u)\ v)$, it is desirable to have u being of type $mexp \to mexp$. For u of type $empty \to mexp$, we would like to define, inside coq, the corresponding $(coer\ u)$ of type $(mexp \to mexp)$. More generally, we look for a coq term $coer$ of type

$$(n:nat)((list\ empty\ n) \to mexp) \to ((list\ mexp\ n) \to mexp)$$

We were not able to construct this term because the Match operator is not powerful enough. Fortunately, we were able to construct the inverse coercion, which is sufficient for our purposes, thanks to higher-order unification.

Now, we would like higher-order unification to operate smoothly when asked for a solution of the equation $< (list\ empty\ n) \to mexp > (coer\ X) = f$, namely we would like it to find the single 'natural' solution, the only one which does not use (more than f) the Match constructor. For that, the only way seems to define a predicate *restr* which rules out the non-natural solutions. The predicate $(restr\ f)$ means that f has no Match operator except possibly in its subtrees of type nat or bool.

[...]

3.3 Semantics

We define here the corresponding type inference rules and evaluation rules of Mini-Ml, using the new middle-order syntax. [...]

4 Top-level

We discuss here some features of the top-level we plan to built over Coq.

4.1 Unicity package. Inverse package

Coq proofs in Natural Semantics seem to be relatively short terms, provided that two 'tools' are given. The first tool we need is the ability to generate some 'intuitive' facts about an -inductive or not- definition of a syntax, in the case where the user asks for them. The intuitive facts about the *exp* syntax simply say that the type exp define trees. With the help of a Match on exp, we could easily prove those facts. But, since in our example, exp is not an inductive definition, we must give them as axioms:

Axiom *uniq_mlpair* :	$(x, y, x', y' : exp)((\langle exp\rangle(mlpair\ x\ y) = (mlpair\ x'\ y'))$
	$\to (((\langle exp\rangle x = x') \wedge (\langle exp\rangle y = y'))$.
Axiom *uniq_arrow* :	$(x, y, x', y' : mltype)((\langle mltype\rangle(arrow\ x\ y) = (arrow\ x'\ y'))$
	$\to (((\langle mltype\rangle x = x') \wedge (\langle mltype\rangle y = y'))$.
Axiom *lam_mlpair* :	$(E : exp \to exp)(x, y : exp)((\langle exp\rangle(lam\ E) \neq (mlpair\ x\ y))$.
Axiom *bool_mlpair* :	$(e : bool)(x, y : exp)((\langle exp\rangle(mlbool\ e) \neq (mlpair\ x\ y))$.
Axiom *nat_mlpair* :	$(e : nat)(x, y : exp)((\langle exp\rangle(mlnat\ e) \neq (mlpair\ x\ y))$.
Axiom *null_mlpair* :	$(x, y : exp)((\langle exp\rangle null \neq (mlpair\ x\ y))$.

The second tool we need is the compilation of the 'inversion' of a set of inference rules, declared as an -inductive or not- definition. In the case of an inductive definition, there is work in progress in the Formel team at Inria-Rocquencourt (Samuel Boutin). The simplest example is the inversion of the *value* judgment, which reduces to the inversion of rule *val_mlpair*:

Theorem *value_inv_mlpair* : $(e1, e2 : exp)(value\ (mlpair\ e1\ e2)) \to ((value\ e1) \wedge (value\ e2))$

As an example, we give in Annex A a proof of the above inversion rule. As another example, the inversion of the Mini-Ml *type* definition is given in AnnexB.

4.2 Proofs in english

Pretty-print from Coq proof terms into proofs written in english, in a more conventional mathematical style. [...]

As an example, let us give a short proof in Coq. Given the previous semantic definitions, we can prove that 'the evaluation of a Mini-Ml expression returns a value':

$$(E, V : exp)(eval\ E\ V) \to (value\ V)$$

The proof proceeds by induction on the length of the proof of $(eval\ E\ V)$. It uses both the unicity and the inverse packages given as an example in the previous subsection.

```
Goal (E,V:exp) (eval E V) → (value V).
Induction 1.
Intro; Apply val_bool.
Intro; Apply val_nat.
Trivial. Trivial.
Apply val_null.
Intro; Apply val_lam.
Trivial.
Intros; Apply val_bool.
Intros; Apply val_bool.
Intros; Apply val_mlpair.Assumption.Assumption.
Intros; Apply proj1 with (value v2); Apply value_inv_mlpair; Assumption.
Intros; Apply proj2 with (value v1); Apply value_inv_mlpair; Assumption.
Trivial.
Save ml_eval_returns_value.
[...]
```

4.3 From higher-order to middle-order and the selection process

Generalisation of the work done on the Mini-Ml example. [...]

4.4 Induction, Match operator

In the definition of the constructor $lam : (exp \to exp) \to exp$, the first occurrence of exp is negative. Such kind of type is not allowed in Coq as an inductive type. The reason for that is that, whenever the user define an inductive type, Coq generates an iteration operator *Match* on that type, together with an induction principle for that type, in a framework which does not naturally extend to that case.

The problem concerning the Match function is that $exp \to exp$ define *all* functions from exp to exp, including those that use a Match. With the help of a Match function, we could build a non-normalisable term, or at least a term which has no counter-part in the model we have in mind. In the case of exp, we could write for example an analogue of Δ (which was given to us by Christine Paulin):

$$d = (lam[x : exp]\ (match\ x\ with \cdots\ (lam\ f) \to (f\ x) \cdots))$$

where $(d\ d)$ rewrites to $(d\ d)$

The problem concerning the induction is that an induction principle is not derivable from such definition, at least in the usual sense. A straightforward induction principle for exp might be written as:

$$\begin{array}{l}\forall P : exp \to Prop \\ (\forall a, b : exp\ (P\ a) \to\ (P\ b) \to\ (P\ (app\ a\ b))) \\ \to (\forall f : exp \to exp\ (\forall a : exp\ (P\ a) \to (P\ (f\ a))) \to (P\ (lam\ f))) \\ \to \cdots \\ \to \forall x : exp\ (P\ x).\end{array}$$

This principle can be written in Coq, but it seems difficult to use it in Coq, because of the 'non-denotational' *lam* case.

We propose here a less readable but more usable principle. This principle originates in the remark that a Mini-Ml expression depends on n variables. The principle is thus defined on an expression depending on a list of expressions of length n (denoted as *(exp n)*):

$$\begin{array}{l}\forall n : nat, \forall P : ((expn) \to exp) \to Prop \\ (\forall n : nat, \forall a, b : ((expn) \to exp)\ (P\ n\ a) \to\ (P\ n\ b) \to\ (P\ n\ \lambda x.(app\ (a\ x)\ (b\ x)))) \\ \to (\forall n : nat, \forall f : (((expn) \to exp) \to ((expn) \to exp)) \\ \quad (P\ (S\ n)\ f) \to (P\ n\ \lambda y.\ (lam\ \lambda x.\ (f\ (cons\ x\ y))))) \\ \to \cdots \\ \to \forall x : ((expn) \to exp)\ (P\ x).\end{array}$$

We have discussed here the problem of induction on the type exp example. The discussion naturally extends to the case of a type representing a judgment. [...]

5 Conclusion

To implement our logical systems in the Calculus of Constructions, we have chosen the encoding defined in the LF framework. However, as it is pointed out in [PMP89], we know that in some cases, as for example the cases from our Mini-Ml example, or simply the ⊃I rule of first order logic:

$$\supset I : [A, B : \mathsf{term}]((\mathsf{true}\ A) \to (\mathsf{true}\ B)) \to (\mathsf{true}\ (A \supset B))$$

this encoding cannot be directly used in an inductive definition. Despite that fact, as in the Elf implementation of Mini-Ml, we would like to use this LF encoding, which seems very natural to us.

Our first idea has been to write a top-level to Coq -or an 'interface' for Coq, rather than modifying it. The basis of this top-level is a new programming style for syntax, which we call *middle-order syntax*. This style is sufficiently higher-order for handling bindings and nevertheless defines object syntaxes as inductive types. The cost for that is that, since the match operator enlarges the syntax with new terms which have to be excluded, we have to write an appropriate selection process. We hope this to be hidden in the top-level. This idea will be developped in the final version of the paper.

Another idea is to add a new operator in the Coq language, called a *restrictive arrow*, that we denote here $\hookrightarrow$. Roughly speaking, $exp \hookrightarrow exp$ will denote those functions from exp to exp that are only built from the constructors of exp. Similarly, $(\mathsf{true}\ A) \hookrightarrow (\mathsf{true}\ B)$ will denote those proofs that are only built from the constructors of *true*. There is work in progress on that subject by Joëlle Despeyroux, Chet Murphy and Frank Pfenning.

In the future, we would like to build proofs in Coq, for example, under Centaur. An interface for Coq is under development in our team. Our aim is to write Natural Semantics definitions in an extended Typol, so as to translate them into Coq terms. A natural continuation of this work will be to see proofs as proof trees in Natural Semantics as well as proof terms in the Calculus of Constructions.

6 Annex A: Proof of an inversion rule

We give here the proof of the following theorem:

Theorem $value\_inv\_mlpair : (e_1, e2 : exp)(value\ (mlpair\ e_1\ e2)) \rightarrow ((value\ e_1) \wedge (value\ e2))$

In this proof, we follow the method suggested in the Coq's user's manual. We first define a suitable constant. Then we prove a general property, from which the desired inversion rules will be derivable as simple proofs, or even just instanciations.

```
Definition inv = [e,e1,e2:exp] (⟨ exp ⟩e=(mlpair e1 e2)) → ((value e1) ∧ (value e2)).

Goal (e,e1,e2:exp) (value e) → (inv e e1 e2).
Induction 1.
Intro; Red; Intro; Absurd (⟨ exp ⟩(mlbool e0)=(mlpair e1 e2)).
Apply bool_mlpair. Assumption.
Intro; Red; Intro; Absurd (⟨ exp ⟩(mlnat e0)=(mlpair e1 e2)).
Apply nat_mlpair. Assumption.
Red. Intro.
Absurd (⟨ exp ⟩null=(mlpair e1 e2)).
Apply null_mlpair. Assumption.
Intro; Red.
Intro; Absurd (⟨ exp ⟩(lam E)=(mlpair e1 e2)).
Apply lam_mlpair; Assumption. Assumption.
Intros; Red; Intro; Cut ((⟨ exp ⟩E1=e1) ∧ (⟨ exp ⟩E2=e2)).
Intro; Apply conj.
Replace e1 with E1. Assumption.
Apply proj1 with ⟨ exp ⟩E2=e2; Assumption.
Replace e2 with E2. Assumption.
Apply proj2 with ⟨ exp ⟩E1=e1; Assumption.
Apply uniq_mlpair; Assumption.
Save value_inv.

Goal (e1,e2:exp)(value (mlpair e1 e2))→((value e1) ∧ (value e2)).
Intros. Cut (⟨ exp ⟩(mlpair e1 e2)=(mlpair e1 e2)).
Change (inv (mlpair e1 e2) e1 e2).
Apply value_inv; Assumption.
Trivial.
Save value_inv_mlpair.
```

7 Annex B: Proof of the Subject Reduction Theorem

We first give the inverse definition of *type*, that we need for the proof of our theorem. The listing of the proof itself follows.

Axiom $type\_inv\_bool$: $(e : bool)(t : mltype)(type\ (mlbool\ e)\ t) \to \langle mltype\rangle t = tbool.$
Axiom $type\_inv\_bool_1$: $(e : bool)(t : mltype)(\langle mltype\rangle t = tbool) \to (type\ (mlbool\ e)\ t).$
Axiom $type\_inv\_nat$: $(e : nat)(t : mltype)(type\ (mlnat\ e)\ t) \to \langle mltype\rangle t = tnat.$
Axiom $type\_inv\_If$: $(e, e_1, e_2 : exp)(t : mltype)(type\ (If\ e\ e_1\ e_2)\ t)$
$\to ((type\ etbool) \wedge (type\ e_1\ t) \wedge (type\ e_2\ t)).$
Axiom $type\_inv\_null$: $(t : mltype)(type\ null\ t) \to \langle mltype\rangle t = (arrow\ tnat\ tbool).$
Axiom $type\_inv\_lam$: $(E : exp \to exp)(t_1, t_2 : mltype)(type\ (lam\ E)\ (arrow\ t_1\ t_2))$
$\to ((x : exp)(type\ x\ t_1) \to (type\ (E\ x)\ t_2)).$
Axiom $type\_inv\_lam_0$: $(E : exp \to exp)(t : mltype)(type\ (lam\ E)\ t)$
$\to \langle mltype\rangle Ex([t_1 : mltype]$
$\langle mltype\rangle Ex([t_2 : mltype](\langle mltype\rangle t = (arrow\ t_1\ t_2)))).$
Axiom $type\_inv\_app$: $(e_1, e_2 : exp)(t_1 : mltype)(type\ (app\ e_1\ e_2)\ t_1)$
$\to \langle mltype\rangle Ex([t_2 : mltype](type\ e_1\ (arrow\ t_2\ t_1)) \wedge (type\ e_2\ t_2)).$
Axiom $type\_inv\_mlpair$: $(e_1, e_2 : exp)(t_1, t_2 : mltype)(type\ (mlpair\ e_1\ e_2)\ (cross\ t_1\ t_2))$
$\to ((type\ e_1\ t_1) \wedge (type\ e_2\ t_2)).$
Axiom $type\_inv\_mlpair_0$: $(e_1, e_2 : exp)(t : mltype)(type\ (mlpair\ e_1\ e_2)\ t)$
$\to \langle mltype\rangle Ex([t_1 : mltype]$
$\langle mltype\rangle Ex([t_2 : mltype](\langle mltype\rangle t = (cross\ t_1\ t_2)))).$
Axiom $type\_inv\_fst$: $(e : exp)(t_1 : mltype)(type\ (fst\ e)\ t_1)$
$\to \langle mltype\rangle Ex([t_2 : mltype](type\ e\ (cross\ t_1\ t_2))).$
Axiom $type\_inv\_snd$: $(e : exp)(t_2 : mltype)(type\ (snd\ e)\ t_2)$
$\to \langle mltype\rangle Ex([t_1 : mltype](type\ e\ (cross\ t_1\ t_2))).$
Axiom $type\_inv\_let$: $(e_1 : exp)(E : exp \to exp)(t : mltype)(type\ (let\ e_1\ E)\ t)$
$\to (\langle mltype\rangle Ex([t0 : mltype](type\ e_1\ t0))$
$\wedge(x : exp)((t_1 : mltype)(type\ e_1\ t_1) \to (type\ x\ t_1))$
$\to (type\ (E\ x)\ t)).$
Axiom $type\_inv\_letrec$: $(E_1, E : exp \to exp)(t : mltype)(type\ (letrec\ E_1\ E)\ t)$
$\to (\langle mltype\rangle Ex([t0 : mltype](type\ (fix\ E_1)\ t0))$
$\wedge(x : exp)((t_1 : mltype)(type\ (fix\ E_1)\ t_1) \to (type\ x\ t_1))$
$\to (type\ (E\ x)\ t)).$
Axiom $type\_inv\_fix$: $(E : exp \to exp)(t : mltype)(type\ (fix\ E)t)$
$\to ((x : exp)(type\ x\ t) \to (type\ (E\ x)\ t)).$

```
Goal (e,v:exp)(eval e v) → (t:mltype)(type e t)→(type v t).
Induction 1.
Trivial. Trivial. Do 4 Intro; Intros ve0 te0 ve1 te1 t tIf; Apply te1;
Apply proj1 with (type e2 t); Apply proj2 with (type e0 tbool);
Apply type_inv_If; Assumption.
Do 4 Intro; Intros ve0 te0 ve2 te2 t tIf; Apply te2;
Apply proj2 with (type e1 t); Apply proj2 with (type e0 tbool);
Apply type_inv_If; Assumption.
Trivial. Trivial. Do 5 Intro; Intros vlam tlam ve2 te2 vApp tApp t tapp; Apply tApp;
Cut ⟨ mltype ⟩Ex([t2:mltype](type e1 (arrow t2 t))∧(type e2 t2)).
Intro Ext2; Elim Ext2; Intro t2; Intro tE1tE2; Apply type_inv_lam with t2.
Apply tlam; Apply proj1 with (type e2 t2); Assumption.
Apply te2; Apply proj2 with (type e1 (arrow t2 t)); Assumption.
Apply type_inv_app; Assumption.
Do 3 Intro; Intro te1; Do 3 Intro; Intro tapp;
Cut ⟨ mltype ⟩Ex([t2:mltype](type e1 (arrow t2 t))∧(type e2 t2)).
Intro Ext2; Elim Ext2; Intro t2; Intro te1e2; Apply type_inv_bool1;
Apply proj2 with ⟨ mltype ⟩t2=tnat; Apply uniq_arrow; Apply type_inv_null; Apply te1;
Apply proj1 with (type e2 t2); Assumption. Apply type_inv_app; Assumption.
Do 4 Intro; Intro te1; Do 3 Intro; Intro tapp;
Cut ⟨ mltype ⟩Ex([t2:mltype](type e1 (arrow t2 t))∧(type e2 t2)).
Intro Ext2; Elim Ext2; Intro t2; Intro te2e1; Apply type_inv_bool1;
Apply proj2 with ⟨ mltype ⟩t2=tnat; Apply uniq_arrow; Apply type_inv_null; Apply te1;
Apply proj1 with (type e2 t2); Assumption. Apply type_inv_app; Assumption.
Do 4 Intro; Intros ve1 te1 ve2 te2 t te1e2;
Cut ⟨ mltype ⟩Ex([t1:mltype]⟨ mltype ⟩Ex([t2:mltype](⟨ mltype ⟩t=(cross t1 t2)))).
Intro Ext1; Elim Ext1; Intro t1; Intro Ext2; Elim Ext2; Intro t2; Intro tc;
Replace t with (cross t1 t2). Apply type_mlpair.
Apply te1;Apply proj1 with (type e2 t2);Apply type_inv_mlpair; Elim tc; Apply te1e2.
Apply te2;Apply proj2 with (type e1 t1);Apply type_inv_mlpair; Elim tc; Apply te1e2.
Apply type_inv_mlpair0 with e1 e2; Assumption.
Do 3 Intro; Intros ve0 te0 t tf; Cut ⟨ mltype ⟩Ex([t2:mltype](type e0 (cross t t2))).
Intro Ext2; Elim Ext2; Intros t2 te0c; Apply proj1 with (type v2 t2);
Apply type_inv_mlpair;Apply te0;Assumption. Apply type_inv_fst; Assumption.
Do 3 Intro; Intros ve0 te0 t ts; Cut ⟨ mltype ⟩Ex([t1:mltype](type e0 (cross t1 t))).
Intro Ext1; Elim Ext1; Intros t1 te0c; Apply proj2 with (type v1 t1);
Apply type_inv_mlpair;Apply te0;Assumption. Apply type_inv_snd; Assumption.
Do 4 Intro; Intros ve1 te1 vApp tApp t tlet; Apply tApp;
Cut ⟨ mltype ⟩Ex([t0:mltype](type e1 t0))∧
(v1:exp)((t1:mltype)(type e1 t1)→(type v1 t1))→(type (E v1) t).
Intro A;Elim A;Intro;Intro B;Apply B;Do 2 Intro;Apply te1;Assumption.
Apply type_inv_let; Assumption.
Do 4 Intro; Intros vfix tfix vApp tApp t tletrec; Apply tApp;
Cut ⟨ mltype ⟩Ex([t0:mltype](type (fix E1) t0))∧
(v1:exp)((t1:mltype)(type (fix E1) t1)→(type v1 t1))→(type (E v1) t).
Intro A;Elim A;Intro;Intro B;Apply B;Do 2 Intro;Apply tfix;Assumption.
Apply type_inv_letrec; Assumption.
Do 2 Intro; Intros vApp tApp t tfix; Apply tApp; Apply type_inv_fix; Assumption.
Save ml_subject_red.
```

References

[AHM87] A. Avron, F. Honsell, and A. Mason. Using typed λ-calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Edinburgh University, Edinburgh, July 1987.

[BCD+88] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the 3rd Symp. on Software Development Environments, November 1988, Boston, USA*, 1988. also available as a Technical Report 777, Inria-Sophia-Antipolis, France, December 1987.

[CDDK86] D. Clement, J. Despeyroux, Th. Despeyroux, and G. Kahn. A simple applicative language: Mini-ml. In *Proceedings of the Symposium on Lisp and Functional Programming*, 1986.

[Des92] J. Despeyroux. Theo: an interactive proof development system. *Scandinavian Journal on Computer Science and Numerical Analysis (BIT), special issue on 'Programming Logic', containing the Proceedings of the Workshop on Programming Logic, Bastad, Sweden, May 21-26 1989*, 32:15–29, 1992. a preliminary version is available as a Research Report RR-887, Inria-Sophia-Antipolis, France, August 1988.

[DFH+91] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin, and B. Werner. The coq proof assistant user's guide, version 5.6. Technical Report 134, Inria, Rocquencourt, France, December 1991.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the POPL ACM Conference on Principles of Programming Languages*, pages 207–212, 1982.

[Har90] R. Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[MP91] S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in elf. In Lars Halln"as, editor, *Proceedings of the Second Workshop on Extentions of Logic Programming*, Springer-Verlag LNCS, 1991. also available as a Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science,Saarbrucken, Germany, August 1991.

[Pfe89] F. Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Proceedings of the fourth ACM-IEEE Symp. on Logic In Computer Science, Asilomar, California, USA*, June 1989.

[PMP89] Ch. Paulin-Mohring and F. Pfenning. Inductively defined types in the calculus of constructions. In *Proceedings of the Fith International Conf. Mathematical Foundations of Programming Semantics, New Orleans, Luisiana, USA*, pages 209–228, 1989.

Universes and a General Notion of Simultaneous Inductive-Recursive Definition in Type Theory (Draft)

Peter Dybjer*
Chalmers University of Technology

August 1992

Abstract

In Martin-Löf's type theory we may define new sets (and families of sets) inductively, and new functions by recursion on the way the elements of these sets are generated. The schema for such definitions that we considered before includes all the standard set formers of type theory with the exception of universes.

Here we give an extended schema which also covers universes à la Tarski. These consist of simultaneous inductive definitions of sets of codes for small sets and recursive definitions of decoding functions. This extension is a small modification of the old schema and includes a general formulation of the notion of a simultaneous inductive-recursive definition.

There are several interesting applications of this extension. Here we show how to obtain an external universe hierarchy, where each level in the hierarchy faithfully reflects the previous level. We also show how to obtain the universe constructions of Griffor and Palmgren. These include an internal unfaithful universe hierarchy and a super universe.

Other examples are the construction of Frege structures in type theory, and various constructions relevant to the formalization of type theory inside type theory.

1 Introduction

The first universe à la Tarski [11] consists of the simultaneous inductive definition of the set U_0 of codes for small sets and the recursive definitions of the decoding function T_0. We have the following rules of *formation* (using Martin-Löf's term)

$$\begin{aligned} U_0 &: set, \\ T_0 &: (U_0)set. \end{aligned}$$

We also have introduction and equality rules for constructors reflecting Π- and Eq-formation:

$$\pi_0 : (u : U_0)(u' : (x : T_0(u))U_0)U_0,$$

$$T_0(\pi_0(u, u')) = \Pi(T_0(u), (x)T_0(u'(x))),$$

$$eq_0 : (u : U_0)(b, b' : T_0(u))U_0,$$

*peterd@cs.chalmers.se

$$T_0(eq_0(u, b, b')) = Eq(T_0(u), b, b').$$

This definition does not fit the schema for inductive and recursive definition in Dybjer [5], since T_0 appears in the introduction rules for U_0. Note that it even appears negatively in the rule for π_0.

In spite of this simultaneity we can argue that it is a *predicative* definition. For example, the rules for π_0 stipulate the following way of constructing new elements of U_0.

At a certain stage we may have constructed the element u. Since T_0 is defined by U_0-recursion, we immediately construct the set $T_0(u)$. Hence it is possible to construct a function u' with domain $T_0(u)$ and range (the presently constructed elements of) U_0. Hence it makes sense to construct an element $\pi_0(u, u')$. Moreover, we can define the $T_0(\pi_0(u, u'))$ in terms of the already constructed sets $T_0(u)$ and $T_0(u'(x))$ for $x : T_0(u)$.

For similar reasons we shall accept a certain general notion of simultaneous inductive-recursive definition. The formulation of this notion will appear as a minor modification of the schema for inductive and recursive definitions in type theory which was presented in Dybjer [5]. The main change is that we allow the simultaneous inductive definition of a family of sets P and the P-recursive definition of a function f.

The idea to consider such a general notion was inspired by Nax Mendler's paper [12] on the category-theoretic semantics of universes in type theory.

Note that this generalization is only possible if we adopt a schematic approach to recursive definitions, since elimination rules only apply to recursive definitions on a previously defined set. For example, T_0 is defined by a recursive schema and not in terms of U_0-elimination.

Moreover, it is also essential that we do not require functions defined by recursion to take their values in a particular set (as in the traditional elimination rules) but allow them to take their values in an arbitrary type. For example, the values may be in the type of sets itself, so we have *recursively defined families of sets.*

So the conceptual priority of certain concepts have been reversed as compared to the standard formulations of Martin-Löf [10, 11]:

elimination rules	$\longleftrightarrow$	recursion schemata
type-valued recursion	$\longleftrightarrow$	universes

Martin-Löf [9] has previously considered a formulation of type theory in terms of recursion schemata. Other arguments for viewing recursive definitions schematically (and allowing more general forms of pattern matching) can be found in Thierry Coquand's paper in this volume [2].

The idea to obtain type-valued recursion in Martin-Löf's type theory by formulating typed-valued ("large") elimination rules rather than by using the ordinary elimination rules in conjunction with universes is due to Bengt Nordström. Smith [9] gave an interpretation of a theory with such large elimination rules in a theory with ordinary elimination rules and universes.

2 Schema for simultaneous inductive-recursive definitions

The reader is referred to Dybjer [5] for a presentation of the old schema, which uses Martin-Löf's theory of logical types. (Alternative presentations can be found in Dybjer [3] of a schema for Martin-Löf's type theory which does not use the theory of logical types, and Coquand and Paulin [2] of a schema for the Calculus of Constructions.) I have tried to use similar notation in order to highlight the similarity between the new and old schemas. In particular there is the notation $a :: \alpha$, meaning that a is a sequence of objects fitting the sequence (relative context, telescope) α of dependent types. The notion of an *s-type* (set-like type) is also essential: it is a type which is built up by the function type construction from base types which are sets. The reason for requiring that certain types occurring in the schema are s-types rather than sets is to cover Martin-Löf's [11, preface] rules for Π.

I present the case with one inductive and one recursive definition simultaneously. There are obvious generalizations to the case with several such simultaneous definitions. Furthermore, I assume that there are no parameters, but the discussion of these in Dybjer [5] can be extended in a straightforward way.

A definition is always relative to a theory containing the rules for previously defined concepts. Thus the requirements on the different parts of the definitions ($\alpha, \psi, \beta, \xi, p, q$ below) are always judgements with respect to that theory.

2.1 Formation rules

Consider introducing a family of sets P together with a function f defined by P-recursion:

$$P : (a :: \alpha)set,$$

$$f : (a :: \alpha)(c : P(a))\psi[a].$$

Here we require that α is a sequence of s-types and $\psi[a]$ type $(a :: \alpha)$.

2.2 Introduction rules

A premise of an introduction rule is either *non-recursive* or *recursive*.

- A non-recursive premise has the form

  $$b : \beta,$$

  where β s-type depending on the previous premises (see below).

- A recursive premise has the form

  $$u : (x :: \xi)P(p[x]),$$

  where ξ is a sequence of s-types, and $p[x] :: \alpha\ (x :: \xi)$ depending on the previous premises (see below). If ξ is empty the premise is *ordinary* and otherwise *generalized* (as in ordinary and generalized induction).

The type of the conclusion of the introduction rule has the form

$$P(q),$$

where $q :: \alpha$ depending on the previous premises (see below).

We write $\beta = \beta[\ldots, b', \ldots, u', \ldots]$, etc. to indicate explicitly the dependence on previous non-recursive premise $b' : \beta'$ and recursive premises $u' : (x' :: \xi')P(p'[x'])$. (This notation is not intended to indicate that non-recursive premises always come before recursive premises.)

We require that

$$\beta[\ldots, b', \ldots, u', \ldots] = \hat{\beta}[\ldots, b', \ldots, (x')f(p'[x'], u'(x')), \ldots],$$

where $\hat{\beta}[\ldots, b', \ldots, v', \ldots]$ s-type $(\ldots, b' : \beta', \ldots, v' : (x' :: \xi')\psi[p'[x']], \ldots)$. Note that the context of $\hat{\beta}$ is obtained from the context (list of previous premises) of β by replacing each recursive premise of the form $u' : (x' :: \xi')P(p'[x'])$ by $v' : (x' :: \xi')\psi[p'[x']]$.

The dependence of ξ, p, and q on previous premises are obtained in the same way as for β.

2.3 Equality rules for the simultaneously defined function

Let *intro* be a constructor constant and $b : \beta$ indicate a typical non-recursive premise and $u : (x :: \xi)P(p[x])$ indicate a typical non-recursive premise of the corresponding introduction rule. The form of the equality rule for f and *intro* is:

$$f(q, \mathit{intro}(\ldots, b, \ldots, u, \ldots)) = e(\ldots, b, \ldots, (x)f(p[x], u(x)), \ldots),$$

where $e(\ldots, b, \ldots, v, \ldots) : \psi[q]\ (\ldots, b : \beta, \ldots, v : (x :: \xi)\psi[p[x]], \ldots)$.

2.4 The analogue of universe elimination

Universe elimination expresses definition by U_0-recursion *after* U_0 and T_0 are defined.

We express this schematically for the general case. Let P and f be defined by a simultaneous inductive-recursive definition as above. We may then define a new function

$$f' : (a :: \alpha)(c : P(a))\psi'[a, c],$$

by P-recursion after P and f have been defined. Here we require that $\psi'[a, c]\ (a :: \alpha, c : P(a))$.

The equality rule has the form

$$f'(q, \mathit{intro}(\ldots, b, \ldots, u, \ldots)) = e'(\ldots, b, \ldots, u, (x)f'(p[x], u(x)), \ldots)$$

where

$$e'(\ldots, b, \ldots, u, v, \ldots) : \psi'[q, \mathit{intro}(\ldots, b, \ldots, u, \ldots)]$$
$$(\ldots, b : \beta, \ldots, u : (x :: \xi)P(p[x]), v : (x :: \xi)\psi'[p[x], u(x)], \ldots).$$

Note that, in contrast to ψ and e in the schema for f, ψ' depends on c as well as on a and e' on u as well as on v in the schema for f'.

3 A simple example: lists with distinct elements

The following example illustrates how the schema can be used. It is a simplification of an example of a simultaneous inductive-recursive definition used by Catarina Coquand in her work (in progress) on formalizing the typed λ-calculus in type theory. She defined the notion of a correct context and the notion of a fresh variable in this way.

Let

$$\begin{array}{rcl} A & : & \mathit{set}, \\ \# & : & (A)(A)\mathit{set} \end{array}$$

be parameters to this definition. A is the set from which the elements of the list are taken and # is an arbitrary relation, which we interpret as "distinctness".

We then simultaneously define the set *Dlist* of lists with distinct elements and the relation *Fresh*, which expresses that a new element is distinct from all elements in a list.

3.1 Formation rules

$$\begin{array}{rcl} \mathit{Dlist} & : & \mathit{set}, \\ \mathit{Fresh} & : & (c : \mathit{Dlist})(a : A)\mathit{set}. \end{array}$$

We see that this is an instance of the schema with α empty and $\psi = (a : A)\mathit{set}$.

3.2 Introduction and equality rules

$$nil : Dlist,$$

$$Fresh(nil) = (a)\top,$$

$$cons : (b : A)(u : Dlist)(b' : Fresh(u, b))Dlist,$$

$$Fresh(cons(b, u)) = (a)(b\#a \wedge Fresh(u, a)).$$

Let us describe in detail how to obtain the rule for *cons* from the schema.

The first premise $b : A$ is non-recursive with $\beta = A$ which is a set and hence an s-type. The second premise is recursive but ordinary, so there is nothing to check. The third premise is non-recursive with

$$\beta[b, u] = Fresh(u, b) = \hat{\beta}[b, Fresh(u)],$$

where $\hat{\beta}[b, v] = v(b)$, which is a set and hence an s-type in the context $(b : A, v : (a : A)set)$. Note that here we have a dependency of a non-recursive premise on the previous (both non-recursive and recursive) premises.

The conclusion has the right form: since α is empty there is nothing more to check.

3.3 A function defined by the analogue of universe elimination

We can define a function which computes the length of a *Dlist* by using an instance of the general schema corresponding to universe elimination:

$$length : (c : Dlist)N.$$

Here $\psi' = N$.

The equality rules are

$$\begin{array}{rcl} length(nil) & = & 0, \\ length(cons(b, u, b')) & = & s(length(u)), \end{array}$$

which easily can be seen to be instances of the general schema.

4 Universes à la Tarski - faithful reflection

4.1 The first universe

We return to the first universe and show how it is obtained by instantiating the schema. The formation rules are

$$\begin{array}{rcl} U_0 & : & set, \\ T_0 & : & (U_0)set. \end{array}$$

Here α is empty and $\psi = set$.

The introduction rule reflecting Π-formation is

$$\pi_0 : (u : U_0)(u' : (x : T_0(u))U_0)U_0.$$

It has a first ordinary recursive premise, and a second generalized recursive premise with $\xi = T_0(u)$. Thus we have an example where a recursive premise depends on an earlier recursive premise.

The corresponding equality rule is

$$T_0(\pi_0(u, u')) = \Pi(T_0(u), (x)T_0(u'(x))).$$

The introduction rule reflecting Eq-formation is

$$eq_0 : (u : U_0)(b, b' : T_0(u))U_0.$$

It has a first ordinary recursive premise, and a second and third non-recursive premise with $\beta = T_0(u)$. Thus we have another example when a non-recursive premise depends on an earlier recursive premise.

The corresponding equality rule is

$$T_0(eq_0(u, b, b')) = Eq(T_0(u), b, b').$$

4.2 The second universe

The formation rules are

$$\begin{array}{rcl} U_1 & : & set, \\ T_1 & : & (U_1)set. \end{array}$$

Here α is empty and $\psi = set$.

There are analogous rules for the constructors π_1 and eq_1 to those of the first universe for reflecting Π and Eq-formation respectively. There is also a rule reflecting U_0-formation.

$$u_{01} : U_1.$$

It has no premise.

The corresponding equality rule is

$$T_1(u_{01}) = U_0$$

Since T_0 is defined by U_0-recursion, it is natural to reflect it as a function

$$t_{01} : (U_0)U_1,$$

which is defined by U_0-recursion:

$$\begin{array}{rcl} t_{01}(\pi_0(u, u')) & = & \pi_1(t_{01}(u), (x)t_{01}(u'(x))), \\ t_{01}(eq_0(u, b, b')) & = & eq_1(t_{01}(u), b, b'). \end{array}$$

This definition comes after the definition of U_0 and T_0, so it is an instance of the schema corresponding to universe-elimination.

This is the "first version" of the universe hierarchy in the paper on "Transfinite hierarchies of universes" in Palmgren [13], since t_{01} is not a constructor. Palmgren motivates the recursion equations for t_{01} by the principle that $T_1(a) = T_1(b)$ should imply that $a = b$, which is also the reason for calling it "faithful reflection".

4.3 Higher universes

We can continue in an analogous way and define U_2 and T_2, U_3 and T_3, etc. Thus for each set A definable in a theory T, there is an extension of T which contains a universe U_n, which contains a code for A. So we get an external universe hierarchy.

5 Unfaithful reflection and its internalization

5.1 The second universe

The "second version" of the second universe in Palmgren [13] treats the code

$$t_{01} : (U_0)U_1$$

for T_0 as a constructor instead of a recursively defined function. Hence we need an extra equality rule for T_1:

$$T_1(t_{01}(b)) = T_0(b)$$

This makes the reflection unfaithful, since in this case different codes may denote the same set.

5.2 The next universe construction

Griffor and Palmgren showed that the "second version" of the construction of U_{n+1} from U_n can be internalized. We introduce the new set formers *Nextu* and *Nextt*, such that $U_{s(n)} = Nextu(U_n, T_n)$ and $T_{s(n)} = Nextt(U_n, T_n)$. This construction also falls under the schema in section 2.

Let

$$\begin{aligned} U &: set, \\ T &: (U)set \end{aligned}$$

be parameters of the definition: we need to construct a universe on top of an arbitrary family U, T.

Formation and typing

$$\begin{aligned} Nextu &: set, \\ Nextt &: (Nextu)set. \end{aligned}$$

Introduction and equality rules correspond to those of the first universe, but we also need to reflect the code set U as

$$* : Nextu,$$

$$Nextt(*) = U$$

and the decoding function T as

$$t : (b : U)Nextu,$$

$$Nextt(t(b)) = T(b).$$

5.3 The super universe

The super universe in Palmgren [13] (this idea was also originally presented in unpublished work together with Ed Griffor) is obtained by reflecting the next-universe construction, viewed as the set formers

$$\begin{aligned} \mathit{Nextu} &: (U : set)(T : (U)set)set, \\ \mathit{Nextt} &: (U : set)(T : (U)set)(\mathit{Nextu}(U,T))set \end{aligned}$$

as well. This super universe also falls under the schema in section 2.

Formation and typing rules

$$\begin{aligned} U_\infty &: set, \\ T_\infty &: (U_\infty)set. \end{aligned}$$

Introduction and equality rules correspond to those for the first universe, but we also need to reflect the first universe U_0

$$u_0 : U_\infty,$$

$$T_\infty(u_0) = U_0$$

and the next-universe construction *Nextu*

$$\mathit{nextu} : (u : U_\infty)(u' : (x : T_\infty(u))U_\infty)U_\infty,$$

$$T_\infty(\mathit{nextu}(u,u')) = \mathit{Nextu}(T_\infty(u),(x)T_\infty(u'(x)))$$

as constructors for U_∞.

As before there is a choice as to reflect T_0 and *Nextt* either faithfully as recursively defined functions or unfaithfully as constructors. The details are similar to the situation for U_1 presented above.

6 Other applications

Other notions which seem to be obtainable by a simultaneous inductive-recursive definition are the following.

- The *propositions and truths* in a Frege structure [1]. In Dybjer [7] this method is used for constructing a Frege structure inside Martin-Löf's type theory.
- The *computability predicates for types and terms* in a theory with dependent types [8].
- The generalization of *logical relations* to the case of dependent types [3].

The details of the two latter constructions are not yet developed.

References

[1] P. Aczel. *Frege Structures and the Notions of Proposition, Truth, and Set*, pages 31–59 in The Kleene Volume. North-Holland, 1980.

[2] T. Coquand. Pattern matching with dependent types. This volume.

[3] T. Coquand. *An Algorithm for Testing Conversion in Type Theory*, pages 255–279 in Logical Frameworks. Cambridge University Press, 1991.

[4] T. Coquand and C. Paulin. Inductively defined types, preliminary version. In *LNCS 417, COLOG '88, International Conference on Computer Logic*. Springer-Verlag, 1990.

[5] P. Dybjer. An inversion principle for Martin-Löf's type theory. In *Proceedings of the Workshop on Programming Logic.* Programming Methodology Group Report 54, Chalmers University of Technology and University of Göteborg", May 1989.

[6] P. Dybjer. *Inductive sets and families in Martin-Löf's Type Theory and their set-theoretic semantics*, pages 280–306 in Logical Frameworks. Cambridge University Press, 1991.

[7] P. Dybjer. Constructing a Frege structure in predicative type theory. Draft paper, 1992.

[8] P. Martin-Löf. An intuitionistic theory of types. Unpublished report, 1972.

[9] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[10] P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[11] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[12] P. F. Mendler. Predicative type universes and primitive recursion. In *Proceedings Sixth Annual Synposium on Logic in Computer Science.* IEEE Computer Society Press, 1991.

[13] E. Palmgren. *On Fixed Point Operators, Inductive Definitions and Universes in Martin-Löf's Type Theory.* PhD thesis, Uppsala University, 1991.

[14] J. Smith. Propositional functions and families of types. *Notre Dame Journal of Formal Logic*, 30(3):442–458, 1989.

Formalizing Properties of Well-Quasi Ordered Sets in ALF

DRAFT

Daniel Fridlender*
University of Göteborg and Chalmers University of Technology

June 1992

Abstract

To prove termination of an algorithm taken from polynomial ideal theory, some general properties of ordered sets must be proved. With the aim of deriving this algorithm in ALF, constructive proofs of these properties and their formalization in ALF are presented.

Introduction

This paper presents some preliminary work on the line of completely formalizing in ALF a constructive proof of a proposition taken from polynomial ideal theory. In the classical proof of this proposition, a classical argument is used to prove the unexistence of infinite sequences of tuples of natural numbers

$$(t_1^1, \ldots, t_n^1), \ldots, (t_1^m, \ldots, t_n^m), \ldots$$

such that

$$\forall i, j \in \mathcal{N}. i < j \Longrightarrow \exists k \in 1 \ldots n. t_k^i \not\leq t_k^j. \tag{1}$$

Next section explains the proposition taken from polynomial ideal theory and the role this classical argument plays in the classical proof. The following section formulates ways of expressing the unexistence of such infinite sequences in a constructive setting. After that, the formalization of two constructive arguments is described. To conclude, we discuss some problems that arose with the formalization and show how they may be solved with the pattern-matching facility of ALF described in [4].

Motivation

The proposition taken from polynomial ideal theory states the existence of Gröbner bases for finitely generated polynomial ideals over discrete fields. More precisely, the proposition says that

> given a field $\mathcal{K}$ with decidable equality, a natural number n, an admissible order $\prec$ over $\mathcal{N}^n$ and a finitely generated polynomial ideal $\mathcal{I}$, there exist polynomials $g_1, \ldots, g_s$ such that $\{g_1, \ldots, g_s\}$ is a Gröbner basis of $\mathcal{I}$ with respect to $\prec$.

*frito@cs.chalmers.se

The field $\mathcal{K}$ is the set from which the coefficients of the polynomials are taken. The number n is the number of indeterminates for the polynomials and so, $\mathcal{N}^n$ is the set of exponents. Given an exponent $e \in \mathcal{N}^n$ and $i \in 1 \dots n$, e_i is the i-th coordinate of e. We define $d(e) = e_1 + \dots + e_n$. The notions of *admissible order* and *Gröbner basis* involved in the proposition are introduced below. After that, we briefly explain how the need of proving the unexistence of infinite sequences satisfying (1) arises.

The reader is referred to [1] and [6] for detailed descriptions of the proposition, and to [2] and [1] for its applications.

Admissible orders

When dealing with polynomials with only one indeterminate, the way to write them is very natural: we use the usual order $<$ over natural numbers and we write the monomials in decreasing order of their exponents. It might be irrelevant when we just want to write them, but it turns out to be important when we want to write a procedure to divide a polynomial by another because it grants the procedure to terminate.

There is not such a natural way (or may be there are many) when we speak of polynomials with many indeterminates. For instance the **lexicographical** order between tuples

$$e \prec_l f \iff \exists j \in 1 \dots n.(e_j < f_j \wedge \forall i \in 1 \dots (j-1).e_i = f_i)$$

or the **degree lexicographical** order

$$e \prec_d f \iff d(e) < d(f) \vee (d(e) = d(f) \wedge e \prec_l f)$$

or the **reverse** order

$$\begin{aligned} e \prec_r f \quad \iff \quad & d(e) < d(f) \vee \\ & \vee (d(e) = d(f) \wedge \exists j \in 1 \dots n.(e_j > f_j \wedge \forall i \in (j+1) \dots n.e_i = f_i)) \end{aligned}$$

are "suitable" ways of determining this "way of writing" polynomials. By suitable we mean that procedures like the division will terminate.

We precise the notion of being "suitable" by defining **admissible orders**. A total order $\prec$ over $\mathcal{N}^n$ is admissible iff

- $\forall e \in \mathcal{N}^n.(0, \dots, 0) \preceq e$
- $\forall e, f, g \in \mathcal{N}^n.e \prec f \Longrightarrow e +^n g \prec f +^n g$

where $+^n$ is the componentwise addition.

Gröbner bases

Consider the case of polynomials with only one indeterminate. To compute the remainder of the division of f by $g = a_n X^n + a_{n-1} X^{n-1} + \dots + a_0$ (where a_n is a nonzero element in the field $\mathcal{K}$) we can consider g as a rewriting rule

$$a_n X^n \longrightarrow -a_{n-1} X^{n-1} - \dots - a_0$$

which transforms polynomials of the form

$$f_1 = p + b_{m+n} X^{m+n} + \dots + b_m X^m + q$$

into

$$f_2 = p + (b_{m+n-1} - \frac{b_{m+n}}{a_n} a_{n-1}) X^{m+n-1} + \ldots + (b_m - \frac{b_{m+n}}{a_n} a_0) X^m + q,$$

where

$$\begin{array}{rcl} p & = & b_{m+n+o} X^{m+n+o} + \ldots + b_{m+n+1} X^{m+n+1} \\ q & = & b_{m-1} X^{m-1} + \ldots + b_0. \end{array}$$

The usual algorithm to compute the remainder r of the division of f by g consists of a sequence of such transformations. It can be seen that any sequence $f_1, \ldots, f_t$ such that

- $f = f_1$,
- $\forall i \in 2 \ldots t$, f_{i-1} is transformed to f_i, and
- f_t can not be further transformed,

produces the remainder of the division of f by g, which is f_t.

Once we have an admissible order this can be generalized to the case of n indeterminates. Given a nonzero polynomial g we define $e_\prec(g)$ to be the greatest exponent of g with respect to $\prec$, $c_\prec(g)$ the coefficient with which it appears in g and $m_\prec(g)$ the monomial $c_\prec(g) X^{e_\prec(g)}$. Notice that we use the notation cX^e for $cX_1^{e_1} \ldots X_n^{e_n}$. We can look at g as a rewriting rule

$$m_\prec(g) \longrightarrow m_\prec(g) - g.$$

If a polynomial f_1 has no monomial m with exponent e such that $e_\prec(g) \leq^n e$ we say that f_1 *is in normal form with respect to g and $\prec$*. In the case that f_1 has a monomial cX^e such that $e_\prec(g) \leq^n e$ we say that f_1 *reduces to f_2 by g and $\prec$ using the monomial of exponent e* and we write it $f_1 \xrightarrow{e,g,\prec} f_2$, where

$$\begin{array}{rcl} f_2 & = & f_1 - \frac{cX^e}{m_\prec(g)} g \\ & = & f_1 - \frac{c}{c_\prec(g)} X^{e -^n e_\prec(g)} g. \end{array}$$

where $-^n$ is the componentwise subtraction.

Now, we can think of having a set of polynomials or rules $\mathcal{G} = \{g_1, \ldots, g_s\}$. By $f_1 \xrightarrow{\mathcal{G},\prec} f_2$ we mean that there is some $e \in \mathcal{N}^n$ and some $g \in \mathcal{G}$ such that $f_1 \xrightarrow{e,g,\prec} f_2$, and we say that f_1 *reduces to f_2 by $\mathcal{G}$ and $\prec$*. The polynomial f *is in normal form with respect to $\mathcal{G}$ and $\prec$* if for all $g \in \mathcal{G}$ it is in normal form with respect to g and $\prec$.

Again we can think of several steps of $\xrightarrow{\mathcal{G},\prec}$ as a division of a polynomial f by a set of polynomials $\mathcal{G}$. The fact of being $\prec$ an admissible order grants the procedure to terminate. But the difference is that now there is not such a unique remainder because in each step we have many possible rules to apply and in general, $\xrightarrow{\mathcal{G},\prec}$ is not confluent. A set $\mathcal{G}$ of polynomials is called a *Gröbner basis with respect to* $\prec$ iff $\xrightarrow{\mathcal{G},\prec}$ is confluent.

There are many other characterizations of Gröbner bases (see [1] and [6]).

Notice that the notion of being a Gröbner basis depends on the admissible order $\prec$ and so, a set may not be a Gröbner basis with respect to an admissible order in spite of being a Gröbner basis with respect to another admissible order.

We say that $\mathcal{G}$ *is a Gröbner basis of an ideal $\mathcal{I}$ with respect to* $\prec$ iff it is a Gröbner basis with respect to $\prec$ and the ideal generated by $\mathcal{G}$ is $\mathcal{I}$.

Classical Proof

The classical proof of the proposition (see [3], [6] and [1]) and consists of:

- A procedure that computes a sequence $g_1, \dots, g_s, \dots$ of polynomials.
- A proof of absurdity from the assumption that the sequence of polynomials computed by the procedure is infinite, that is, a classical proof of termination.

To look at the classical proof with more detail, we define the relation $<^n$ over $\mathcal{N}^n$ by

$$e \leq^n f \Longleftrightarrow \forall i \in 1 \dots n. e_i \leq f_i$$

With this the classical proof consists of:

- A procedure that computes a sequence $g_1, \dots, g_s, \dots$ of nonzero polynomials with greatest exponents $e^1, \dots, e^s, \dots$ (i.e., $e_i = e_{\prec}(g_i)$) such that

$$\forall i, j \in \mathcal{N}. i < j \Longrightarrow e^i \not\leq^n e^j. \tag{2}$$

- A proof of absurdity from the assumption that an infinite sequence of exponents satisfying (2) exists.

The second point, which is the classical proof of termination, states exactly the unexistence of an infinite sequence satisfying (1), because of the equivalence between (1) and (2). A sequence $e^1, \dots, e^s, \dots$ (finite or infinite) satisfying (2) is called a *bad sequence.*

Thus, in what follows, we will formalize constructive proofs of the unexistence of *infinite* bad sequences.

Constructive Formulations

Two approaches have been taken to express termination in a constructive way.

The first one was inspired in the fact that the infinite sequence $e^1, \dots, e^s, \dots$ is *bad* iff the **finite** sequences $[], [e^1], \dots, [e^1, \dots, e^s], \dots$ are all *bad sequences.* Thus, instead of speaking of the unexistence of *infinite* bad sequences, we can speak of the well-foundedness of the relation $\sqsubset$ on *finite* bad sequences defined by

$$[e^1, \dots, e^{s_1}] \sqsubset [f^1, \dots, f^{s_2}] \Longleftrightarrow s_1 = s_2 + 1 \land \forall i \in 1 \dots s_2. e^i = f^i.$$

The second approach is dual to this one and is essentially the one chosen in [7]. A sequence $e^1, \dots, e^s, \dots$ is *good* iff there exist $i, j \in \mathcal{N}$ such that $i < j$ and $e^i \leq^n e^j$. The decidability of $\leq^n$ implies that the sets $\mathcal{B}$ and $\mathcal{G}$ of all the finite *bad* and *good* sequences respectively, form a partition of the set of finite sequences. Thus, instead of speaking of the unexistence of *infinite bad sequences*, we can speak of the existence of a *good* initial segment of any infinite sequence. We follow [9] expressing this universal quantification over infinite sequences by means of bar induction.

Thus, given a set $\mathcal{A}$ and a relation $<$ on it, we will have two different definitions of the predicate *well*[1]. According to the first one, $<$ *is well* iff the relation $\sqsubset$ defined by

$$[a_1, \dots, a_n] \sqsubset [b_1, \dots, b_m] \Longleftrightarrow n = m + 1 \land \forall i \in 1 \dots m. a_i = b_i$$

[1] The properties we are interested in, hold for any *well* relation. This is why we define the notion of *well* instead of the more known notion of *well quasi-order* suggested by the title of this paper

is well-founded on the set of *finite bad sequences* (of elements of $\mathcal{A}$). According to the second one, $<$ *is well* iff the predicate *to be good*, on finite sequences of elements of $\mathcal{A}$, is a bar.

In both approaches the strategy taken was to prove as a lemma that if $<_\mathcal{A}$ and $<_\mathcal{B}$ are *well* on $\mathcal{A}$ and $\mathcal{B}$ respectively, so is $<_{\mathcal{A}\times\mathcal{B}}$ on $\mathcal{A}\times\mathcal{B}$, where

$$(a,b) \leq_{\mathcal{A}\times\mathcal{B}} (c,d) \iff a \leq_\mathcal{A} c \wedge b \leq_\mathcal{B} d.$$

This, together with a proof that $<$ (the usual order on natural numbers) is *well* will give a proof that $<^2$ is *well* on $\mathcal{N}^2$, and by induction, that $<^n$ is *well* on $\mathcal{N}^n$ for any natural number n.

Thus, in spite of being interested only in the relation $<^n$, we proved the lemmas in a more general level involving any relation on any set.

Formalizations

The formalizetions were done in ALF (see [8]). The new features for making definitions in contexts were used extensively. We briefly show the syntax for dealing with contexts and substitutions.

To give a name to a context:

$$\texttt{name} \quad = \quad \texttt{context}$$

To make definitions in a context we show as an example the definition of *tsil* (lists in reverse order) where the context is `[A:Set]`:

$$\begin{array}{lll l}
\texttt{tsil} & : & \texttt{Set} & [\texttt{A}:\texttt{Set}] \\
[] & : & \texttt{tsil} & [\texttt{A}:\texttt{Set}] \\
. & : & (\texttt{tsil};\texttt{A})\texttt{tsil} & [\texttt{A}:\texttt{Set}] \\
\texttt{tsilrec} & : & (\texttt{P}:(\texttt{tsil})\texttt{Set}; & \\
 & & \texttt{P}([]); & \\
 & & (\texttt{l}:\texttt{tsil};\texttt{a}:\texttt{A};\texttt{P}(\texttt{l}))\texttt{P}(\texttt{l.a}); & \\
 & & \texttt{l}:\texttt{tsil})\texttt{P}(\texttt{l}) & [\texttt{A}:\texttt{Set}]
\end{array}$$

$$\begin{array}{lll}
\texttt{tsilrec}(\texttt{P},\texttt{d},\texttt{e},[]) & = & \texttt{d} \\
\texttt{tsilrec}(\texttt{P},\texttt{d},\texttt{e},\texttt{l.a}) & = & \texttt{e}(\texttt{l},\texttt{a},\texttt{tsilrec}(\texttt{P},\texttt{d},\texttt{e},\texttt{l}))
\end{array}$$

Substitutions can also be given a name:

$$\texttt{name} \quad = \quad \{\texttt{x}_1 := \texttt{t}_1, \ldots, \texttt{x}_\texttt{n} := \texttt{t}_\texttt{n}\} \quad \texttt{C}_1 \quad \texttt{C}_2$$

where the context $\texttt{C}_1$ contains a typing for $\texttt{x}_1, \ldots, \texttt{x}_\texttt{n}$ and the free variables in $\texttt{t}_1, \ldots, \texttt{t}_\texttt{n}$ are typed in the context $\texttt{C}_2$. The syntax of the application of a substitution σ to a term $\texttt{t}$ is $\texttt{t}\{\sigma\}$.

In the definitions that follow we use contexts to represent the structure we work with: a relation on a set.

First Approach

With the definition of *tsil* above, we say that an order $<$ on a set $\mathcal{A}$ *is well* iff the relation $\sqsubset$ defined by

$$l \sqsubset k \iff \exists a \in \mathcal{A}. l = k.a$$

is well-founded on the *subset* of the *bad tsils*. Thus, the definition of well-foundedness must be defined not just for a relation, but for a relation on a *subset*.

In the proof that $<_{\mathcal{A}\times\mathcal{B}}$ is *well* on $\mathcal{A}\times\mathcal{B}$ provided that $<_{\mathcal{A}}$ and $<_{\mathcal{B}}$ are *well* on $\mathcal{A}$ and $\mathcal{B}$ respectively, we followed [10] where this result is proved for *well quasi-orders* (*well* relations that are quasi-orders). The main lemma in [10] is that $<$ is *well* on $\mathcal{A}$ iff for all $a \in \mathcal{A}$ its restriction is *well* on $\mathcal{A}[a]$, which is defined there as a *subset* of $\mathcal{A}$.

These arguments led us to define relations with a context having the relation itself, the set on which it is defined and the *subset* on which the behavior of the relation is interesting:

$$\mathtt{REL} \quad = \quad [\mathtt{A}:\mathtt{Set};\mathtt{As}:(\mathtt{A})\mathtt{Set};<:(\mathtt{A};\mathtt{A})\mathtt{Set}] \ .$$

We define for a relation to be well-founded, as it is frequently done in constructive mathematics. We define the *accessible part of a set by a relation*, slightly modified here to consider the case of a relation on a *subset*:

$$\begin{array}{lll}
\mathtt{acc} & : & (\mathtt{A})\mathtt{Set} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{REL}\\
\mathtt{acc_I} & : & (\mathtt{a}:\mathtt{A};\mathtt{As}(\mathtt{a});(\mathtt{b}:\mathtt{A};\mathtt{As}(\mathtt{b});\mathtt{b}<\mathtt{a})\mathtt{acc}(\mathtt{b}))\mathtt{acc}(\mathtt{a}) \qquad\qquad \mathtt{REL}\\
\mathtt{acc_E} & : & (\mathtt{P}:(\mathtt{a}:\mathtt{A};\mathtt{acc}(\mathtt{a}))\mathtt{Set};\\
& & \ (\mathtt{a}:\mathtt{A};\\
& & \ \mathtt{as}:\mathtt{As}(\mathtt{a});\\
& & \ \mathtt{q}:(\mathtt{b}:\mathtt{A};\mathtt{As}(\mathtt{b});\mathtt{b}<\mathtt{a})\mathtt{acc}(\mathtt{b});\\
& & \ (\mathtt{b}:\mathtt{A};\mathtt{bs}:\mathtt{As}(\mathtt{b});\mathtt{r}:\mathtt{b}<\mathtt{a})\mathtt{P}(\mathtt{b},\mathtt{q}(\mathtt{b},\mathtt{bs},\mathtt{r})))\mathtt{P}(\mathtt{a},\mathtt{acc_I}(\mathtt{a},\mathtt{as},\mathtt{q}));\\
& & \mathtt{a}:\mathtt{A};\\
& & \mathtt{p}:\mathtt{acc}(\mathtt{a}))\mathtt{P}(\mathtt{a},\mathtt{p}) \qquad\qquad\qquad\qquad\qquad\qquad \mathtt{REL}
\end{array}$$

$$\mathtt{acc_E}(\mathtt{P},\mathtt{d},\mathtt{a},\mathtt{acc_I}(\mathtt{a},\mathtt{as},\mathtt{q})) \quad = \quad \mathtt{d}(\mathtt{a},\mathtt{as},\mathtt{q},[\mathtt{b},\mathtt{bs},\mathtt{r}]\mathtt{acc_E}(\mathtt{P},\mathtt{d},\mathtt{b},\mathtt{q}(\mathtt{b},\mathtt{bs},\mathtt{r})))$$

Finally, a relation is well-founded if all the elements of the subset are in the accessible part:

$$\mathtt{wf} \quad = \quad \forall \mathtt{a} \in \mathtt{A}.\mathtt{As}(\mathtt{a}) \supset \mathtt{acc}(\mathtt{a}) \quad : \quad \mathtt{Set} \quad \mathtt{REL}$$

We informally said that a relation is *well* iff the relation $\sqsubset$ defined above, is well-founded on the subset of *bad tsils*:

$$\mathtt{well} \quad = \quad \mathtt{wf}\{\mathtt{A}:=\mathtt{tsil};\mathtt{As}:=\mathtt{bad};<:=\sqsubset\} \quad : \quad \mathtt{Set} \quad \mathtt{REL}$$

The set `tsil` was already defined. To define the subset (or predicate) `bad` a more elementary notion seems to be necessary: the one of being a *bad element for a tsil*. Given a relation $<$ on $\mathcal{A}$, a *tsil* $[a_1,\ldots,a_s]$ and $a \in \mathcal{A}$, a is a *bad element* for $[a_1,\ldots,a_s]$ iff

$$\forall i \in 1 \ldots s . a_i \not\leq a$$

where $\leq$ is the reflexive closure of $<$ and $\mathtt{not} = [\mathtt{A}]\mathtt{A} \supset \bot : (\mathtt{Set})\mathtt{Set}$.

Two possible definitions of *bad element* have been analized: by an inductive definition, or by an implicit definition. We have chosen the latter, because the former, though being more elegant and clear, seems not to be strong enough to prove some properties. Some comments on this are presented in the last part of this paper.

$$\mathtt{badelem} \quad : \quad (\mathtt{tsil};\mathtt{A})\mathtt{Set} \quad \mathtt{REL}$$

$$\begin{array}{lcl} \mathtt{badelem}([],\mathtt{a}) & = & \mathtt{As}(\mathtt{a}) \\ \mathtt{badelem}(\mathtt{l.b},\mathtt{a}) & = & \mathtt{badelem}(\mathtt{l},\mathtt{a}) \wedge \mathtt{b} \not\leq \mathtt{a} \end{array}$$

The same arguments led us to use implicit definitions to define bad and $\sqsubset$:

$$\mathtt{bad} \ : \ (\mathtt{tsil})\mathtt{Set} \quad \mathtt{REL}$$

$$\begin{array}{lcl} \mathtt{bad}([]) & = & \top \\ \mathtt{bad}(\mathtt{l.a}) & = & \mathtt{bad}(\mathtt{l}) \wedge \mathtt{badelem}(\mathtt{l},\mathtt{a}) \end{array}$$

$$\sqsubset \ : \ (\mathtt{tsil};\mathtt{tsil})\mathtt{Set} \quad \mathtt{REL}$$

$$\begin{array}{lcl} [] \sqsubset \mathtt{l} & = & \bot \\ \mathtt{l.b} \sqsubset \mathtt{l}' & = & \mathtt{l} =_{\mathtt{tsil}} \mathtt{l}' \end{array}$$

In [10] given a relation $<$ on $\mathcal{A}$ and a finite sequence $[a_1, \ldots, a_n]$ the subset of $\mathcal{A}$, $\mathcal{A}[a_1, \ldots, a_s]$ is defined by

$$\mathcal{A}[a_1, \ldots, a_s] = \{a \in \mathcal{A} | \forall i \in 1 \ldots s. a_i \not\leq a\}.$$

The main lemma there is that $<$ is *well* on $\mathcal{A}$ iff for all $a \in \mathcal{A}$, it is *well* on $\mathcal{A}[a]$. This is stated in ALF with two lemmas:

$$\begin{array}{lclcll} \mathtt{lemma}_1 & = & ? & : & (\mathtt{well};\mathtt{l}:\mathtt{tsil})\mathtt{well}\{\mathtt{As} := \mathtt{badelem}(\mathtt{l})\} & \mathtt{REL} \\ \mathtt{lemma}_2 & = & ? & : & ((\mathtt{a}:\mathtt{A})\mathtt{well}\{\mathtt{As} := \mathtt{badelem}([].\mathtt{a})\})\mathtt{well} & \mathtt{REL} \end{array}$$

being lemma 1 a more general statement than the “only if” part of the main lemma.

What follows presents a formalization of the statements proved in [10]. Lemma 3 states that if a relation is *well* on a subset, it is also *well* on a subset of it; lemma 4, that if a relation is *well* on two subsets, it is well on their union.

$$\begin{array}{lclcll} \mathtt{lemma}_3 & = & ? & : & (\mathtt{Bs}:(\mathtt{A})\mathtt{Set};(\mathtt{a}:\mathtt{A};\mathtt{Bs}(\mathtt{a}))\mathtt{As}(\mathtt{a});\mathtt{well})\mathtt{well}\{\mathtt{As} := \mathtt{Bs}\} & \mathtt{REL} \\ \mathtt{lemma}_4 & = & ? & : & (\mathtt{Bs}:(\mathtt{A})\mathtt{Set}; & \\ & & & & \mathtt{well}; & \\ & & & & \mathtt{well}\{\mathtt{As} := \mathtt{Bs}\})\mathtt{well}\{\mathtt{As} := [\mathtt{a}]\mathtt{As}(\mathtt{a}) \vee \mathtt{Bs}(\mathtt{a})\} & \mathtt{REL} \end{array}$$

The theorem, saying that if $<$ is a decidable relation *well* on $\mathcal{A}$ and $\prec$ is *well* on $\mathcal{B}$, $<_{\mathcal{A}\times\mathcal{B}}$ is *well* on $\mathcal{A} \times \mathcal{B}$.

$$\mathtt{theorem} \ = \ ? \ : \ ((\mathtt{a},\mathtt{b}:\mathtt{A})\mathtt{a} < \mathtt{b} \vee \mathtt{a} \not< \mathtt{b};\mathtt{well};\mathtt{well}_{\mathtt{B}})\mathtt{well}\{\sigma\} \quad \mathtt{REL}+\mathtt{REL}_{\mathtt{B}}$$

where the substitution σ is

$$\begin{array}{lcl} \sigma & = & \{\mathtt{A} := \mathtt{A} \wedge \mathtt{B}; \\ & & \mathtt{As} := [\mathtt{p}]\mathtt{As}(\mathtt{fst}(\mathtt{p})) \wedge \mathtt{Bs}(\mathtt{snd}(\mathtt{p})); \\ & & < := [\mathtt{p},\mathtt{q}] <_{\mathcal{A}\times\mathcal{B}} (\mathtt{fst}(\mathtt{p}),\mathtt{snd}(\mathtt{p}),\mathtt{fst}(\mathtt{q}),\mathtt{snd}(\mathtt{q}))\} \ : \ \mathtt{REL} \quad \mathtt{REL}+\mathtt{REL}_{\mathtt{B}} \end{array}.$$

and

$$\begin{array}{lcll} \mathtt{REL}_{\mathtt{B}} & = & [\mathtt{B}:\mathtt{Set};\mathtt{Bs}:(\mathtt{B})\mathtt{Set};\prec:(\mathtt{B};\mathtt{B})\mathtt{Set}] & \\ \mathtt{well}_{\mathtt{B}} & = & \mathtt{well}\{\mathtt{A} := \mathtt{B};\mathtt{As} := \mathtt{Bs};< := \prec\} \quad : \ \mathtt{Set} & \mathtt{REL}_{\mathtt{B}} \\ <_{\mathcal{A}\times\mathcal{B}} & = & [\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}](\mathtt{a} < \mathtt{c} \wedge \mathtt{b} =_{\mathtt{B}} \mathtt{d}) & \\ & & \vee (\mathtt{a} =_{\mathtt{A}} \mathtt{c} \wedge \mathtt{b} \prec \mathtt{d}) & \\ & & \vee (\mathtt{a} < \mathtt{c} \wedge \mathtt{b} \prec \mathtt{d}) \quad : \ (\mathtt{A};\mathtt{B};\mathtt{A};\mathtt{B})\mathtt{Set} & \mathtt{REL}+\mathtt{REL}_{\mathtt{B}} \end{array}$$

The formal proof follows the arguments in [10]. The formalization is not difficult but has many details that make it laborious.

Second Approach

Contrary to what happened in the first approach, now there is no need to consider subsets. The notion of *well* is now defined as a property that the whole set of *tsils* has to satisfy, which is, that the predicate *good* is a bar. Thus, the structure we need is represented by the context:

$$\texttt{REL} \;=\; [\texttt{A} : \texttt{Set}; <: (\texttt{A};\texttt{A})\texttt{Set}] \;.$$

We define the dual notion to *bad*, *to be good*, which uses a previous notion of being a *good element*. Given a relation $<$ on $\mathcal{A}$, a *tsil* $[a_1, \ldots, a_s]$ and $a \in \mathcal{A}$, a is a *good element* for $[a_1, \ldots, a_s]$ iff

$$\exists i \in 1 \ldots s . a_i \leq a$$

This time we chose an inductive definition because at the time this proof was done, the new facilities with pattern-matching were already under development and they solved the weakness of this definition:

$$\begin{array}{lll}
\texttt{goodelem} & : & (\texttt{tsil};\texttt{A})\texttt{Set} \qquad\qquad\qquad\qquad\qquad \texttt{REL} \\
\texttt{goodelem}_{\texttt{I1}} & : & (\texttt{l}:\texttt{tsil};\texttt{a},\texttt{b}:\texttt{A};\texttt{a}\leq\texttt{b})\texttt{goodelem}(\texttt{l.a},\texttt{b}) \qquad \texttt{REL} \\
\texttt{goodelem}_{\texttt{I2}} & : & (\texttt{l}:\texttt{tsil};\texttt{a},\texttt{b}:\texttt{A};\texttt{goodelem}(\texttt{l},\texttt{b}))\texttt{goodelem}(\texttt{l.a},\texttt{b}) \qquad \texttt{REL} \\
\texttt{goodelem}_{\texttt{E}} & : & (\texttt{b}:\texttt{A}; \\
 & & \texttt{P}:(\texttt{l}:\texttt{tsil};\texttt{goodelem}(\texttt{l},\texttt{b}))\texttt{Set}; \\
 & & (\texttt{l}:\texttt{tsil};\texttt{a}:\texttt{A};\texttt{r}:\texttt{a}\leq\texttt{b})\texttt{P}(\texttt{l.a},\texttt{goodelem}_{\texttt{I1}}(\texttt{l},\texttt{a},\texttt{b},\texttt{r})); \\
 & & \;(\texttt{l}:\texttt{tsil}; \\
 & & \;\texttt{a}:\texttt{A}; \\
 & & \;\texttt{g}:\texttt{goodelem}(\texttt{l},\texttt{b}); \\
 & & \;\texttt{P}(\texttt{l},\texttt{g}))\texttt{P}(\texttt{l.a},\texttt{goodelem}_{\texttt{I2}}(\texttt{l},\texttt{a},\texttt{b},\texttt{g})); \\
 & & \texttt{l}:\texttt{tsil}; \\
 & & \texttt{g}:\texttt{goodelem}(\texttt{l},\texttt{b}))\texttt{P}(\texttt{l},\texttt{g}) \qquad\qquad\qquad \texttt{REL}
\end{array}$$

$$\begin{array}{lll}
\texttt{goodelem}_{\texttt{E}}(\texttt{b},\texttt{P},\texttt{d},\texttt{e},\texttt{l.a},\texttt{goodelem}_{\texttt{I1}}(\texttt{l},\texttt{a},\texttt{b},\texttt{r})) & = & \texttt{d}(\texttt{l},\texttt{a},\texttt{r}) \\
\texttt{goodelem}_{\texttt{E}}(\texttt{b},\texttt{P},\texttt{d},\texttt{e},\texttt{l.a},\texttt{goodelem}_{\texttt{I2}}(\texttt{l},\texttt{a},\texttt{b},\texttt{g})) & = & \texttt{e}(\texttt{l},\texttt{a},\texttt{g},\texttt{goodelem}_{\texttt{E}}(\texttt{b},\texttt{P},\texttt{d},\texttt{e},\texttt{l},\texttt{g}))
\end{array}$$

The same argument led us to use inductive definitions for the predicate *good*.

$$\begin{array}{lll}
\texttt{good} & : & (\texttt{tsil})\texttt{Set} \qquad\qquad\qquad\qquad\qquad \texttt{REL} \\
\texttt{good}_{\texttt{I1}} & : & (\texttt{l}:\texttt{tsil};\texttt{a}:\texttt{A};\texttt{goodelem}(\texttt{l},\texttt{a}))\texttt{good}(\texttt{l.a}) \qquad \texttt{REL} \\
\texttt{good}_{\texttt{I2}} & : & (\texttt{l}:\texttt{tsil};\texttt{a}:\texttt{A};\texttt{good}(\texttt{l}))\texttt{good}(\texttt{l.a}) \qquad \texttt{REL} \\
\texttt{good}_{\texttt{E}} & : & (\texttt{P}:(\texttt{l}:\texttt{tsil};\texttt{good}(\texttt{l}))\texttt{Set}; \\
 & & (\texttt{l}:\texttt{tsil};\texttt{a}:\texttt{A};\texttt{g}:\texttt{goodelem}(\texttt{l},\texttt{a}))\texttt{P}(\texttt{l.a},\texttt{good}_{\texttt{I1}}(\texttt{l},\texttt{a},\texttt{g})); \\
 & & (\texttt{l}:\texttt{tsil};\texttt{a}:\texttt{A};\texttt{g}:\texttt{good}(\texttt{l});\texttt{P}(\texttt{l},\texttt{g}))\texttt{P}(\texttt{l.a},\texttt{good}_{\texttt{I2}}(\texttt{l},\texttt{a},\texttt{g})); \\
 & & \texttt{l}:\texttt{tsil}; \\
 & & \texttt{g}:\texttt{good}(\texttt{l}))\texttt{P}(\texttt{l},\texttt{g}) \qquad\qquad\qquad \texttt{REL}
\end{array}$$

$$\begin{array}{lll}
\texttt{good}_{\texttt{E}}(\texttt{P},\texttt{d},\texttt{e},\texttt{l.a},\texttt{good}_{\texttt{I1}}(\texttt{l},\texttt{a},\texttt{g})) & = & \texttt{d}(\texttt{l},\texttt{a},\texttt{g}) \\
\texttt{good}_{\texttt{E}}(\texttt{P},\texttt{d},\texttt{e},\texttt{l.a},\texttt{good}_{\texttt{I2}}(\texttt{l},\texttt{a},\texttt{g})) & = & \texttt{e}(\texttt{l},\texttt{a},\texttt{g},\texttt{good}_{\texttt{E}}(\texttt{P},\texttt{d},\texttt{e},\texttt{l},\texttt{g}))
\end{array}$$

With this approach we express the unexistence of *infinite* bad sequences by saying that every infinite sequence has an initial segment which is *good*. Following [9] we express this universal quantification over infinite sequences through the notion of *being a bar*. We say that a predicate

P on *tsils bars* a *tsil* l, either when P(l) holds or when for all a in A it *bars* l.a. For an analysis of bar induction as a constructive principle, we refer to [5].

This is formalized in ALF by the inductive definition:

$$
\begin{array}{lll}
\mathtt{bar} & : & ((\mathtt{tsil})\mathtt{Set};\mathtt{tsil})\mathtt{Set} \quad [\mathtt{A}:\mathtt{Set}] \\
\mathtt{bar}_{\mathtt{I1}} & : & (\mathtt{P}:(\mathtt{tsil})\mathtt{Set};\mathtt{l}:\mathtt{tsil};\mathtt{P}(\mathtt{l}))\mathtt{bar}(\mathtt{P},\mathtt{l}) \quad [\mathtt{A}:\mathtt{Set}] \\
\mathtt{bar}_{\mathtt{I2}} & : & (\mathtt{P}:(\mathtt{tsil})\mathtt{Set};\mathtt{l}:\mathtt{tsil};(\mathtt{a}:\mathtt{A})\mathtt{bar}(\mathtt{P},\mathtt{l.a}))\mathtt{bar}(\mathtt{P},\mathtt{l}) \quad [\mathtt{A}:\mathtt{Set}] \\
\mathtt{bar}_{\mathtt{E}} & : & (\mathtt{P}:(\mathtt{tsil})\mathtt{Set}; \\
 & & \mathtt{Q}:(\mathtt{l}:\mathtt{tsil};\mathtt{bar}(\mathtt{P},\mathtt{l}))\mathtt{Set}; \\
 & & (\mathtt{l}:\mathtt{tsil};\mathtt{p}:\mathtt{P}(\mathtt{l}))\mathtt{Q}(\mathtt{l},\mathtt{bar}_{\mathtt{I1}}(\mathtt{P},\mathtt{l},\mathtt{p})) \\
 & & \quad (\mathtt{l}:\mathtt{tsil}; \\
 & & \quad \mathtt{q}:(\mathtt{a}:\mathtt{A})\mathtt{bar}(\mathtt{P},\mathtt{l.a}); \\
 & & \quad (\mathtt{a}:\mathtt{A})\mathtt{Q}(\mathtt{l.a},\mathtt{q}(\mathtt{a})))\mathtt{Q}(\mathtt{l},\mathtt{bar}_{\mathtt{I2}}(\mathtt{P},\mathtt{l},\mathtt{q})); \\
 & & \mathtt{l}:\mathtt{tsil}; \\
 & & \mathtt{p}:\mathtt{bar}(\mathtt{P},\mathtt{l}))\mathtt{Q}(\mathtt{l},\mathtt{p}) \quad [\mathtt{A}:\mathtt{Set}]
\end{array}
$$

$$
\begin{array}{lll}
\mathtt{bar}_{\mathtt{E}}(\mathtt{P},\mathtt{Q},\mathtt{d},\mathtt{e},\mathtt{l},\mathtt{bar}_{\mathtt{I1}}(\mathtt{P},\mathtt{l},\mathtt{p})) & = & \mathtt{d}(\mathtt{l},\mathtt{p}) \\
\mathtt{bar}_{\mathtt{E}}(\mathtt{P},\mathtt{Q},\mathtt{d},\mathtt{e},\mathtt{l},\mathtt{bar}_{\mathtt{I2}}(\mathtt{P},\mathtt{l},\mathtt{p})) & = & \mathtt{e}(\mathtt{l},\mathtt{p},[\mathtt{a}]\mathtt{bar}_{\mathtt{E}}(\mathtt{P},\mathtt{Q},\mathtt{d},\mathtt{e},\mathtt{l.a},\mathtt{p}(\mathtt{a}))))
\end{array}
$$

With this, we say that a relation is *well* when *good* bars the empty *tsil*:

$$\mathtt{well} \quad = \quad \mathtt{bar}(\mathtt{good},[]) \quad \mathtt{REL}$$

The proof that if two relation $<$ and $\prec$ are *well* on $\mathcal{A}$ and $\mathcal{B}$ respectively, so is their product on $\mathcal{A} \times \mathcal{B}$ proceeds by defining a 3-ary predicate $\mathcal{P}$ whose arguments are *tsils* of elements of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{A} \times \mathcal{B}$ respectively. Informally, we define $\mathcal{P}$ by

$$
\begin{gathered}
\mathcal{P}(([a_1,\ldots,a_n],[b_1,\ldots,b_m],[(c_1,d_1),\ldots,(c_l,d_l)]) \\
\Longleftrightarrow \\
good([a_1,\ldots,a_n]) \vee \exists i \in 1\ldots n.\exists j \in 1\ldots l.a_i \leq c_j \vee \\
\vee\, good\{\sigma_B\}([b_1,\ldots,b_m]) \vee \exists i \in 1\ldots m.\exists j \in 1\ldots l.b_i \preceq d_j \vee \\
\vee\, good\{\sigma\}([(c_1,d_1),\ldots,(c_l,d_l)])
\end{gathered}
$$

where the definition of σ and σ_B are the intuitive ones.

With this, what is proved is that if $<$ and $\prec$ are *well* on $\mathcal{A}$ and $\mathcal{B}$ respectively, $\mathcal{P}([],[])$ is a bar (on the sequences of pairs). What remains to prove is that *good*([]) does *not* hold.

Some Comments on the Formalizations

In both approaches some inductive definitions seemed to be too weak for our purposes. In the first one, an inductive definition of *bad element* would not allow us to prove $a \not\leq b$ from the fact of being b a *bad element* for a sequence whose last element is a, which is intuitive from the informal definition of being a *bad element*. The difficulty comes from the impossibility of defining the function *last* on *tsils* (or *head* on *lists*). An analogous problem arises in the second approach, in which an inductive definition of *good* is not strong enough to prove that the empty *tsil* is not *good*.

In both problems, the difficulty comes from having to consider in the proofs, while using the elimination rule of the inductive definition, *impossible cases*. The pattern-matching feature presented in [4] eliminates from the case analysis these impossible cases.

References

[1] Bruno Buchberger. *Gröbner Bases: An Algorithmic Method in Polynomial Ideal Theory.* In N.K.Bose, editor, Multidimensional Systems Theory, pages 184-232. D.Reidel Publishing Company. Dordrecht-Boston-Lancaster, 1985.

[2] Bruno Buchberger. *Applications of Gröbner Bases in Non-linear Computational Geometry.* In Proc. Workshop on Scientific Software, IMA, Minneapolis, USA. Springer, 1987.

[3] Bruno Buchberger. *Ein algorithmus zum Auffinden der Basiselemente des Restklasseringes nach einem nulldimensionalen Polynomideal.* Ph. D. Thesis, Univ. Innsbruck, 1965.

[4] Thierry Coquand. *Pattern-matching with dependent types.* These Proceedings.

[5] Michael Dummett. *Elements of Intuitionism.* Oxford, 1977.

[6] Monique Lejeune-Jalabert. *Effectivité de calculs polynomiaux.* D.E.A. Thesis, Inst. Fourier, Univ. Grenoble I, 1985.

[7] Carl Jacobsson, Clas Löfwall. *Standard bases for general coefficient rings and a new constructive proof of Hilbert's basis theorem.* Journal of Symbolic Computation 1991, Vol 12, pg. 337 - 371.

[8] Lena Magnusson. These Proceedings.

[9] Per Martin-Löf. *Notes on Constructive Mathematics.* Almqvist & Wiksell, Stockholm, 1968.

[10] Fred Richman and Gabriel Stolzenberg. *Well Quasi-Ordered Sets.* To appear in Advances in Mathematics.

Formal Proofs of Combinatorial Completeness

Veronica Gaspes*
University of Gothemburg and Chalmers University of Technology

June 1992

Abstract

Formal proofs of the combinatorial completeness of two combinator calculi are presented. The formal language used to write the calculi and the proofs in is Martin-Löf's set theory. The formalization was machine checked in Alf (Another Logical Framework) where the set theory and the notions involved in the proofs were implemented. The first proof concerns untyped combinators where unrestricted combinatorial completeness holds. The presentation of the calculus as well as the proof of combinatorial completeness are from Curry and Feys' book Combinatory Logic [CF58]. The second proof concerns simply typed combinators where a typed restricted version of combinatorial completeness holds.

An appendix is attached with the implementation in Alf of the formalization.

1 Introduction

This paper describes formalization of proofs of some theorems involving combinators. The theorems state the ability of combinators to describe functions without the use of variables. To express these statements, known as combinatorial completeness, the notion of intuitive function is formulated so that it can be stated that any such function can be defined by means of the combinators. First an untyped version of the combinators is considered, taken from Curry and Feys' book Combinatory Logic [CF58]; the definitions and the proofs follow the ones presented there and have been formalized in Martin-Löf's set theory and machine checked using the Alf system. Next a typed version, with types formed from arbitrary atoms by $\rightarrow$, is presented and a type restricted version of the combinatorial completeness is proved. Two approaches to the typed calculus are analyzed. Curry's one, where arbitrary combinators may be formed which are then assigned types; and Church's one where combinators are defined as belonging to some type (this approach is used in the papers by Tait [Tai67] and Sanchís [San67]).

The work was intended as an exercise in developing a concrete example in Martin-Löf's set theory. A description of this theory may be found in the book Programming in Martin-Löf's Type Theory by Nordström, Petersson and Smith [NPS90], from which we borrow the notational conventions. The implementation of the logical framework provided by Alf was of crucial importance in actually completing this development. It provides a straightfoward implementation of the set theory and the sets that are added for the particular problem, and, in checking the definitions and proofs, it helps you by keeping track of a lot of information. A description of an earlier version of Alf may be found in [ACN90]. The set theory was used by defining new sets that correspond to this particular problem. These definitions usualy introduce

*vero@cs.chalmers.se

new primitive notions which are provided by making an inductive definition, but there are also some abbreviations which are stated using definitional equality.

All the informal definitions, statements and proofs are to be found in the literature. They are presented here only as an introduction to the formalization, which then is interleaved with the text and is presented as soon as the notions are introduced. The paper begins with some references to the presentation of formal systems as given in the first chapters of [CF58], emphasizing the relation of that approach to the formalization of theories in Martin-Löf's set theory.

1.1 Formal Systems

The aim of this section is to show some connections between the way formal systems are analyzed in chapters 1 and 2 in [CF58] and the way one formalizes theories in Martin-Löf's set theory. In [CF58] a language, the U-language, and a theory, the epi-theory, are described which are assumed to be understood and are used to comunicate notions and to prove statements. In this context, the description of a formal system consists of extending the U-language in a particular way which also extends the theory in the sense that it provides it with more principles of proof.

A formal system is described by a collection of sets: a set of objects introduced by the formal system, a set of statements or formulae formulated by the formal system (called the elementary statements which could be understood as the forms of judgement introduced by the formal system) and a set of theorems proved by the formal system (called the elementary theorems which may be understood as the evident judgements). The U-language is thus enriched with the new objects and predicate formers. All these sets are introduced by inductive definitions, and hence the epi-theory is enriched with principles of proof by induction over these sets. The principles that arise from different inductive definitions are carefully explained in [CF58].

It is required that the epi-theory is constructive, so that proofs of elementary theorems or construction of elementary objects can be produced from the proof of an epi-theorem stating their existence. If one understands the epi-theory and the U-language as Martin-Löf's set theory, then the description of a formal system consists of extending the set theory with some sets. The extended theory may be used to prove elementary statements, i.e. to use the formal system defined; or epi-statements, i.e. to prove statements about the formal system defined. The sets that characterize the formal system are given by inductive definitions; there are rules to introduce the elements of each set and an elimination rule that expresses the principle of proof or definition by induction over that set. The explanation of the induction techniques in [CF58] may be used as a justification of the elimination rules. Most of the epi-theorems in the book are then proved by induction which can easily be expressed in set theory.

This approach differs from the more usual one in the presentation of formal systems [Kle52], described as more syntactical in [CF58]. For this other view, the first step in the setting of a formal system consists of defining a new language, as formed by the strings over a certain alphabet. This new language is then called the object language and is never used. The language in which the formulation is made is called the metalanguage where there are names for the expressions of the object language. All the use that is done of the formal system introduced is through the metalanguage. In the more abstract approach above, objects are introduced by an inductive definition thus being formed through specific constructors, not merely by sequencing of characters. This approach is followed also in the definition of the forms of judgements and the set of theorems, where the constructors correspond to the axioms and rules of inference. In this case the objects introduced are used, thus constituting an extension of the used language.

This corresponds exactly to the way one introduces new sets in Martin-Löf's set theory, one has then an extended language and theory to use.

1.2 Variables

When considering the definition of a formal system as extending a language and a theory with some constants and rules, it remains to be said how variables are to be handled. The original, primitive language will contain variables, in [CF58] called the intuitive variables, which, in setting up a formal system, are used to express generality, for instance when describing a schematic rule or stating some epi-theorems. It may be the case that the formal system introduced needs to describe functional objects. This is usually done by introducing a category of objects, the formal variables, for which a rule of substitution is formulated. These special objects are then introduced as new constants that extend the primitive language. The aim of combinatory logic is to provide a formal system that analyzes the role of variables and substitution in such a way that it may be used in the presentation of other formal systems when describing functional objects. Hence, once combinatory logic has been introduced, neither variables nor substitution rules need to be considered.

The theory of combinators itself does not contain variables. It will be the case, however, that to state the theorems about the combinatorial completeness (epi-theorems concerning the formal systems we will present for the combinators), we will have to consider formal systems which do contain variables, namely extensions with variables of the original theory. In these extensions the objects will be understood as functions from objects of the original theory to objects of the original theory. These variables added in the extensions will be introduced by only saying that they are objects, no rule of substitution will be stated for them nor will they take part of any statement but as arbitrary objects. In [CF58] these formal variables are called indeterminates. They have the property that they may be replaced by an arbitrary object and thus, the statements involving them are equally valid as general epi-statements using an intuitive variable instead. The theories for combinators will be formalized in Martin-Löf's set theory by defining new sets for the objects, statements and theorems; the variables of an extension will be implemented by a particular set.

2 Untyped Combinators

The theory of combinators provides an analysis of the process of substitution through the notions of function and application. In Schönfinkel's original paper [Sch24], a theory of functional objects is introduced to represent incomplete objects without using variables to indicate the places for the argument. These functional objects, called combinators, are obtained from certain atomic combinators by means of application as the only operation. Substitution is represented by application and there are rules that say which object results when the substitution is performed. The atomic combinators and the rules that define them express the basic ways of building expressions which contain holes, namely, building an arbitrary constant expression (not depending in what is provided as argument) and the incomplete expression resulting from application of an incomplete expression to another.

Following [CF58], the theory of combinators, called $\mathcal{H}$, is defined as a formal system in the sense above, by inductively defining the objects, the statements and the theorems. The objects are formed from the atoms K, to express constant functions, and S, to express the function which results from the application of a function to another one. Application will be denoted by juxtaposition fa, associating to the left. The statements are of the form $f \sim g$, where f and g are arbitrary objects. The theorems are defined by the following axioms and rules

$$Kab \sim a \qquad (K)$$

$$Sfga \sim fa(ga) \qquad (S)$$

$$f \sim f \qquad (\rho)$$

$$\frac{f \sim g}{g \sim f} \qquad (\sigma)$$

$$\frac{f \sim g \quad g \sim h}{f \sim h} \qquad (\tau)$$

$$\frac{f \sim g}{fa \sim ga} \qquad (\mu)$$

$$\frac{a \sim b}{fa \sim fb} \qquad (\nu)$$

As an example of an object of this theory, consider SKK, which, according to the rules, behaves like the identity function, i.e. the function representing the incomplete object which when completed with any other object as argument produces the same object as result: let x be an arbitray object, then

$$SKKx \sim Kx(Kx) \sim x$$

The result we will analyze is the ability of this calculus of functions to describe as an object any intuitive incomplete object. These incomplete objects are understood to be described as a combination of previously defined functions and variables. It will be shown that any such object is already present as an object (a function) of the calculus. This property is known as combinatorial completeness and we need to introduce some way of describing the intuitive incomplete objects in order to state it more precisely. This result makes it possible to work with this calculus, instead of using variables and introducing substitution, when defining other formal systems. We will follow Curry's formulation of intuitive incomplete objects and of what it means for them to be described by a function in the calculus.

A possible way of understanding the intuitively definable functions from objects to objects is by considering combinations formed not only from K and S but involving also certain arguments $x_1, \ldots, x_n$. This may be done by considering a theory similar to the original one where certain unspecified objects $x_1, \ldots, x_n$ have been added. If we let $\mathcal{H}$' be the theory that results by adjoining the indeterminates $x_1, \ldots, x_n$ to $\mathcal{H}$, then an object h' in $\mathcal{H}$' can be understood as an incomplete object which when completed with objects $a_1, \ldots, a_n$ of $\mathcal{H}$ yields an object of $\mathcal{H}$. Thus, what must be shown is that for every h' an object h of $\mathcal{H}$ may be defined so that it represents the incomplete object h' as a function from objects in $\mathcal{H}$ to objects in $\mathcal{H}$. The way of proving that this term h represents h' as a function is done by showing that

$$hx_1 \ldots x_n \sim h'$$

is a theorem of $\mathcal{H}$'. This means, because h does not contain any of the indeterminates $x_1, \ldots, x_n$, that for every tuple $a_1, \ldots, a_n$ of objects of $\mathcal{H}$, the application $ha_1 \ldots a_n$ is equal to the object that results by replacing the indeterminates in h' by $a_1, \ldots, a_n$. Before showing how to define h in $\mathcal{H}$ from h' in $\mathcal{H}$', we will show how to formalize, in Martin-Löf's set theory, the theory $\mathcal{H}$ and its extensions.

To proceed, the notion of extension by adjoining indeterminates must be formulated more precisely. While a theory is characterized by the set of objects, the set of statements and the set of theorems, an extension is defined by the set of indeterminates adjoined. Thus, a theory and its extensions may be described as a family of theories indexed by the sets of indeterminates, i.e. by a family of objects, a family of statements and a family of theorems, all indexed by the sets of indeterminates. All theories in the family are different, however there is a natural way of identifying an arbitrary object of a theory with the one built in the same way in an extension. This is made explicit by injections of a theory to an extended one. When considering this, the statement about $hx_1 \ldots x_n {\sim} h'$ must be formulated more precisely considering these injections: because h is a term in $\mathcal{H}$ it must be injected into $\mathcal{H}$' for the application to $x_1, \ldots, x_n$ to be an element in the corresponding set of objects and the statement to make sense.

The formalization consists of defining sets and families of sets in the set theory to represent the notions being formalized. The following is a presentation of such a formalization for the families of theories of combinators above:

- Each set of indeterminates is a finite set, hence it will be represented by the finite sets of Martin-Löf's theory N_n. These sets will be defined as a family indexed by the length n. The introduction rules for this family correspond to a finite set being formed by injecting all the elements in the previous one and putting one element more. There is no introduction rule for the finite set of length $\mathsf{0}$.

  IS-formation

  $$\mathsf{IS}(n)\ \mathit{Set}\ [n \in \mathsf{N}]$$

  IS-introduction-1

  $$\frac{n \in \mathsf{N}}{\mathsf{put}(n) \in \mathsf{IS}(\mathsf{s}(n))}$$

  IS-introduction-2

  $$\frac{n \in \mathsf{N} \quad x \in \mathsf{IS}(n)}{\mathsf{inject}(n, x) \in \mathsf{IS}(\mathsf{s}(n))}$$

  An elimination rule is given for each set $\mathsf{IS}(\mathrm{n})$ which allows proof of statements under the assumption of having an element in the corresponding segment. The set $\mathsf{IS}(\mathsf{0})$ has no introduction rule, thus its elimination looks like $\perp$-elimination:

IS-elimination-0

$$\frac{P(x)\ \mathit{Set}\ [x \in \mathsf{IS}(0)] \quad p \in \mathsf{IS}(0)}{\mathsf{IS0}E(P,p) \in P(p)}$$

The $\mathsf{IS}(\mathsf{s}(\mathrm{n}))$ has the two introduction rules above, thus,

IS-elimination-s

$$\frac{\begin{array}{l} n \in \mathsf{N} \\ P(x)\ \mathit{Set}\ [x \in \mathsf{IS}(\mathsf{s}(n))] \\ e \in P(\mathsf{put}(n)) \\ d(q) \in P(\mathsf{inject}(n,q))\ [q \in \mathsf{IS}(n)] \\ p \in \mathsf{IS}(\mathsf{s}(n)) \end{array}}{\mathsf{ISs}E(n,P,e,d,p) \in P(p)}$$

This constant is defined by the equalities

$$\mathsf{ISs}E(n,P,e,d,\mathsf{put}(n)) = e$$
$$\mathsf{ISs}E(n,P,e,d,\mathit{Inject}(n,q)) = d(q)$$

- The family Ob indexed by the sizes of the initial segments represents the family of objects indexed by the extensions:

  Ob-formation

$$\mathsf{Ob}(n)\ \mathit{Set}\ [n \in \mathsf{N}]$$

The introduction rules correspond to the formation of the objects in an extension with n indeterminates:

Ob-introduction-1

$$\frac{n \in \mathsf{N}}{\mathsf{K}(n) \in \mathsf{Ob}(n)}$$

Ob-introduction-2

$$\frac{n \in \mathsf{N}}{\mathsf{S}(n) \in \mathsf{Ob}(n)}$$

Ob-introduction-3

$$\frac{n \in \mathsf{N} \quad x \in \mathsf{IS}(n)}{\mathsf{Var}(x,n) \in \mathsf{Ob}(n)}$$

(corresponding to the variable x in an extension with n variables)

Ob-introduction-4

$$\frac{n \in \mathsf{N} \quad f,a \in \mathsf{Ob}(n)}{\mathsf{App}(f,a,n) \in \mathsf{Ob}(n)}$$

There is an elimination rule for the set of objects in each extension:

Ob-elimination

$$\frac{\begin{array}{l} n \in \mathsf{N} \\ P(x)\ \mathit{Set}\ [x \in \mathsf{Ob}(n)] \\ k \in P(\mathsf{K}(n)) \\ s \in P(\mathsf{S}(n)) \\ v(y) \in P(\mathsf{Var}(y,n))\ [y \in \mathsf{IS}(n)] \\ a(h,z,ih,iz) \in P(\mathsf{App}(h,z,n))\ [h,z \in \mathsf{Ob}(n), ih \in P(h), iz \in P(z)] \\ o \in \mathsf{Ob}(n) \end{array}}{\mathsf{Ob}E(n,P,k,s,v,a,o) \in P(o)}$$

This constant is defined by the equalities

$$\begin{array}{l} \mathsf{Ob}E(n,P,k,s,v,a,\mathsf{K}(n)) = k \\ \mathsf{Ob}E(n,P,k,s,v,a,\mathsf{S}(n)) = s \\ \mathsf{Ob}E(n,P,k,s,v,a,\mathsf{Var}(y,n)) = v(y) \\ \mathsf{Ob}E(n,P,k,s,v,a,(\mathsf{App}(h,z,n)) = a(h,z, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Ob}E(n,P,k,s,v,a,h), \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{Ob}E(n,P,k,s,v,a,z)) \end{array}$$

- The family Form of the statements or formulae:

Form-formation

$$\mathsf{Form}(n)\ \mathit{Set}\ [n \in \mathsf{N}]$$

There is only one kind of statement, equality between objects:

Form-introduction

$$\frac{n \in \mathsf{N} \quad a,b \in \mathsf{Ob}(n)}{a \sim_n b \in \mathsf{Form}(n)}$$

Form-elimination

$$\frac{\begin{array}{l} n \in \mathsf{N} \\ P(x)\ \mathit{Set}\ [x \in \mathsf{Form}(n)] \\ e(a,b) \in P(a \sim_n b)\ [a,b \in \mathsf{Ob}(n)] \\ f \in \mathsf{Form}(n) \end{array}}{\mathsf{Form}E(n,P,e,f) \in P(f)}$$

$\mathsf{Form}E(n,P,e,a \sim_n b) = e(a,b)$

- The family $\vdash$ of the theorems:

  $\vdash$-formation

$$\vdash_n f \; Set \; [n \in \mathsf{N}, f \in \mathsf{Form}(n)]$$

  The introduction rules correspond to the axioms and inference rules for asserting equality between terms. Only some of them are exhibited here, the ones corresponding to (K), (ρ) and (μ); the rest are coded in the same way.

  $\vdash$-introduction-K

$$\frac{n \in \mathsf{N} \quad a, b \in \mathsf{Ob}(n)}{\mathsf{kk}(a, b, n) \in \vdash_n \mathsf{App}(\mathsf{App}(\mathsf{K}, a, n), b, n) \sim_n a}$$

  $\vdash$-introduction-ρ

$$\frac{n \in \mathsf{N} \quad a \in \mathsf{Ob}(n)}{\rho(a, n) \in \vdash_n a \sim_n a}$$

  $\vdash$-introduction-μ

$$\frac{n \in \mathsf{N} \quad f, g, a \in \mathsf{Ob}(n) \quad p \in \vdash_n f \sim_n g}{\mu(f, g, a, n, p) \in \vdash_n \mathsf{App}(f, a, n) \sim_n \mathsf{App}(g, a, n)}$$

We have thus defined the sets that describe a family of theories, each set of each theory defined inductively. The elimination rules which close the definitions are used whenever definition by recursion or proof by induction on a given set is required.

There is a natural way of understanding an object of a theory as an object of an extension, built in the same way but in the extended theory. When formalizing the theorem about combinatorial completeness this injection is required and thus must be stated formally. This is done by means of a function

$$\mathcal{O} \in (\Pi m, n \in \mathsf{N})(m \leq n) \supset \mathsf{Ob}(m) \supset \mathsf{Ob}(n)$$

which given m, n with $m \leq n$ will take an object in an extension with m indeterminates into "itself" in an extension with n indeterminates. The definition of $\mathcal{O}$ is based on the injection of an initial segment of length m into the one of length n. To implement this definitions $\leq$ will be defined by first defining $<$ as follows:

$<$-formation

$$m < n \; Set \; [m, n \in \mathsf{N}]$$

The introduction rules state that $<$ is the transitive closure of the relation that a number is less than its succesor:

$<$-introduction-1

$$\frac{n \in \mathsf{N}}{\mathsf{OneStep}(n) \in n < \mathsf{s}(n)}$$

<-introduction-2

$$\frac{m, n, o \in \mathsf{N} \quad m < n \quad n < o}{\tau(m,n,o) \in m < o}$$

The elimination rule closes the definition by stating the principle of proof or definition which allows us to use the knowledge that a number is less than another one according to this definition:

<-elimination

$$\frac{\begin{array}{l} P(x,y,z)\ \mathit{Set}\ [x,y \in \mathsf{N}, z \in m < n] \\ e(x) \in P(x, \mathsf{s}(x), \mathsf{OneStep}(x))\ [x \in \mathsf{N}] \\ d(s,t,u,q,r,i,h) \in P(s,u,\tau(s,t,u)) \left[\begin{array}{l} s,t,u \in \mathsf{N}, \\ q \in s < t, \\ r \in t < u, \\ i \in P(s,t,q), \\ h \in P(t,u,r) \end{array}\right] \\ m, n \in \mathsf{N} \\ p \in m < n \end{array}}{<\!E(P,e,d,m,n,p) \in P(m,n,p)}$$

$$<\!E(P,e,d,m,\mathsf{s}(m),\mathsf{OneStep}(m)) = e(m)$$

$$<\!E(P,e,d,m,o,\tau(m,n,o,q,r)) = d(m,n,o,q,r,<E(P,e,d,m,n,q),<E(P,e,d,n,o,r))$$

With this, the definition of $\mathcal{O}$ is by cases on $m \leq n \equiv m = n \vee m < n$. If $m = n$ it is the identity. If $m < n$ it follows by induction on this fact, by using the elimination rule for the set $<$. The two cases that must be considered are

n is the succesor of m. By induction on $\mathsf{Ob}(m)$ each case of object formation is considered and translated to the corresponding element in $\mathsf{Ob}(n)$,

there is a number o such that $m < o$ and $o < n$. $\mathcal{O}$ is defined to be the composition of the functions the induction hypothesis yields for $m < o$ and $o < n$.

In a similar way the formulae of a theory may be injected as formulae of an extension by means of a function

$$\mathcal{F} \in (\Pi m, n \in \mathsf{N})(m \leq n) \supset \mathsf{Form}(m) \supset \mathsf{Form}(n)$$

which is defined in the same way as $\mathcal{O}$.

2.1 Definition of Abstraction

We now define, for any object h' in $\mathcal{H}$', an object h in $\mathcal{H}$ that represents it as a function from objects in $\mathcal{H}$ to objects in $\mathcal{H}$, and prove this fact. The definition and the proof are by induction on the size of the extension, the inductive step being based on the definition and proof corresponding to passing from a theory with at least one indeterminate to one with the extra indeterminate removed. In order to describe the definition we will use the notation of bracket abstraction for the object defined: if h' is an object in $\mathcal{H}$', we shall write $[x_1, \ldots, x_n]h'$ for the h in $\mathcal{H}$. With this notation the induction step in the definition may be expressed as

$$[x_1,\ \ldots,\ x_n, x]h' \equiv [x_1,\ \ldots,\ x_n]\,([x]\,h')$$

The definition of $[x]h'$ is by induction on h'. For each way of building objects in an extension containing x, it is analyzed what function of x is intended and so an object which represents such a function in a theory without x is defined. The following cases arise:

- h' is K. When thought of as a function of x from objects of $\mathcal{H}$ to objects of $\mathcal{H}$, it is the constant function with value K. Thus we define $[x]K$ to be KK.
- h' is S. With the same argument as for K we define $[x]S$ to be KS.
- h' is a variable of the extension. Then there are two cases to consider; either the variable is x or some other variable y. In the case it is y, the same argument as for K and S, yields the definition of $[x]y$ to be Ky. In the case it is x the definition is such that the function is the identity function. This is because x is the identity function when considered a function of x. Thus, for this case we define $[x]x$ to be SKK which was shown above to behave as the identity.
- h' is an application fg. Both f and g may contain x and thus are to be considered as functions of x. The induction hypothesis yields the definitions of the two functions $[x]f$ and $[x]g$. The object that results when performing the substitution of an object of $\mathcal{H}$ for x in fg may be analyzed as formed by the application of the object that results from performing the substitution in f to the one that results from performing it in g. In terms of $[x]f$ and $[x]g$ this is expressed by the application of $([x]f)o$ to $([x]g)o$ where o is the object substituted for x. We thus define $[x](fg)$ to be $S\,([x]f)\,([x]g)$.

The formalization of these definitions proceeds by first defining $[x]h'$ by induction on the construction of h', an object in a theory that contains at least one variable, i.e. by applying the elimination rule for Ob to $h' \in \mathsf{Ob}(\mathsf{s}(n))$ for some natural number n. Thus, by assuming that one has such an h', an object in $\mathsf{Ob}(n)$ is defined as the abstraction of the extra variable from h'. The cases that arise in the induction on h' are the following:

- h' is $\mathsf{K}(\mathsf{s}(n))$. Then define the abstraction as the constant function with value K in $\mathsf{Ob}(n)$: $\mathsf{App}(\mathsf{K}(n), \mathsf{K}(n), n)$.
- h' is $\mathsf{S}(\mathsf{s}(n))$. Then define the abstraction to be the constant function with value S in $\mathsf{Ob}(n)$: $\mathsf{App}(\mathsf{K}(n), \mathsf{S}(n), n)$.
- h' is $\mathsf{Var}(x, \mathsf{s}(n))$ where $x \in \mathsf{IS}(\mathsf{s}(n))$. Then one must analyze whether x is the variable to be abstracted (the variable that extends the theory from n to $\mathsf{s}(n)$) or any other variable (already present in the theory for n). This is realized by making an induction on x, i.e. by applying the elimination rule for IS to $x \in \mathsf{IS}(\mathsf{s}(n))$. The two cases that arise are
  - x is $\mathsf{put}(n)$. Then define the abstraction to be the identity function in $\mathsf{Ob}(n)$: $\mathsf{App}(\mathsf{App}(\mathsf{S}(n), \mathsf{K}(n), n)\mathsf{K}(n), n)$.
  - x is $\mathsf{inject}(y, n)$ where y is in $\mathsf{IS}(n)$. Then define the abstraction to be the constant function with value y in $\mathsf{Ob}(n)$: $\mathsf{App}(\mathsf{K}(n), \mathsf{Var}(y, n), n)$.

- h' is an application $\mathsf{App}(f, a, \mathsf{s}(n))$. Then the induction hypothesis yields a value for f in $\mathsf{Ob}(n)$, say fh, and a value for a in $\mathsf{Ob}(n)$, say ah, then define the abstraction to be $\mathsf{App}(\mathsf{App}(\mathsf{S}(n), fh, n), ah, n)$.

We have thus defined an element in $\mathsf{Ob}(n)$ assuming $n \in \mathsf{N}$ and h' in $\mathsf{Ob}(\mathsf{s}(n))$. We will now define, by induction on n, an element in $\mathsf{Ob}(0)$ from an element h' in an extension with n indeterminates, the abstraction of all the variables in the extension. Using induction on n two cases arise

- If n is 0, then h' is already an element in $\mathsf{Ob}(0)$.
- If n is the succesor of some natural number t, the induction hypothesis yields a method for defining an element in $\mathsf{Ob}(0)$ for every h in $\mathsf{Ob}(t)$. Then take the element in $\mathsf{Ob}(t)$ which results from appliyng the definition above and finally apply the method yielded by the induction hypothesis to this element.

One must now prove that the abstractions so defined behave as planned, i.e. that $hx_1 \ldots x_n\sim h'$. This proof follows the same lines as the definition of h; again it is carried out by induction on n, the crucial case being the proof that $([x]h')x\sim h'$, which again is an induction on the construction of h':

- h' is K, S, or a variable y different from x (the definition of $[x]h'$ is the same for these three cases, Kh'). Then we have $([x]h')\ x \equiv Kh'x\sim h'$.
- h' is the variable x. Then we have $([x]x)x \equiv SKKx\sim x$.
- h' is the application fg. Then we have

  $$([x](fg))x \ \equiv S\,([x]f)\,([x]g)\,x \sim (([x]f)x)\,(([x]g)x)$$

  The induction hypothesis yields $([x]f)x \sim f$ and $([x]g)x \sim g$ and hence, by application of the rules μ, ν and τ,

  $$(([x]f)x)\,(([x]g)x) \sim fg$$

With the definitions for the theories of combinators above, and letting h' be the object in an extension with n indeterminates and h be $[x_1 \ldots x_n]h'$ in the original theory (an extension with no indeterminates), the fact that h represents h' as a function is formalized by:

$$\vdash_n app^*(\mathcal{O}(h), n) \sim_n h'$$

where $app^*(x, n)$ is defined by induction on n as the application of $x \in \mathsf{Ob}(n)$ to all the indeterminates of the extension. The injection of h into the extension with n indeterminates is needed for the application to $x_1, \ldots, x_n$ to make sense. As in the informal proof above this is shown by induction on n, the induction step being based on the proof of

$$\vdash_{\mathsf{s}(n)} \mathsf{App}(\mathcal{O}(h), \mathsf{put}(n), n) \sim_{\mathsf{s}(n)} h'$$

where h' is an object in an extension with $\mathsf{s}(n)$ indeterminates and h is the corresponding $[x]h'$ in an extension with n indeterminates, the abstracted variable being the last element of the initial segment $\mathsf{IS}(\mathsf{s}(n))$, i.e. $\mathsf{put}(n)$. The induction on h' used in the informal proof is implemented by an application of the elimination rule for $\mathsf{Ob}(n)$. The $\equiv$ that are said to hold in the informal proof hold by definitional equality, while the $\sim$ require the application of the rules that define $\vdash_n$.

The definition of $[x_1 \ldots x_n]h'$ together with the proof of $\vdash_n app^*(\mathcal{O}(h), n) \sim_n h'$ constitute a proof of

$$(\Pi n \in \mathsf{N})(\Pi h' \in \mathsf{Ob}(n))(\Sigma h \in \mathsf{Ob}(0)).\vdash_n app^*(\mathcal{O}(h), n) \sim_n h'$$

which formalizes the combinatorial completeness for untyped combinators. This formalization of the theories and the proofs was edited in Alf, the code of this implementation is attached as an appendix.

3 Typed Combinators

In this section we shall introduce a theory of typed combinators and show that a restricted version of combinatorial completeness holds. After a short motivation for the introduction of types, the theory and the notion of extension by adjoining indeterminates will be presented together with a formalization in Martin-Löf's set theory. Next, a typed version of the combinatorial completeness will be proved, followed by the corresponding formalization.

Different analysis of combinatory logic as a formalism to express the process of substitution in formalized theories end in the introduction of types to classify combinators in different categories. In Curry and Feys [CF58] the types are associated with semantical categories. The coding of propositional logic as an extension of combinatory logic with a constant P for forming implication (Pfg codes $f \supset g$), a new statement constructor $\longmapsto$ for derivability ($\longmapsto f$ codes f is derivable) and rules for proving that certain objects are derivable ($\longmapsto Pf(Pgf)$ codes the axiom $f \supset g \supset f$) results in an inconsistent theory in the sense that for every object f, it is a theorem that $\longmapsto f$. This inconsistency is avoided by restricting the arguments of $\longmapsto$ to be propositions. This implies classifying the objects to isolate the category of propositions. This gives rise to the theory of functionality presented in [CF58] where, from the type of propositions, a functional hierarchy is built. Martin-Löf presents an argument for classifying the combinators in syntactical categories. The untyped combinators are rejected as a basis for the presentation of formal systems due to the fact that equality between them is not decidable and thus cannot be used as definitional equality; and that abbreviations may not be eliminable (consider the term $SII(SII)$). The solution comes from analyzing expressions into saturated and unsaturated as Frege [Fre84] did. This is expressed by means of types, starting from a type for saturated expressions and building a functional hierarchy from it for the unsaturated expressions according to the kind of argument they expect. This system is presented in chapter 3 of [NPS90] for the λ-calculus.

Different notions of meaningfullness led Curry and Church to different ways of considering the typed versions of their calculi. Curry's approach consists of assigning types to the untyped objects while Church's consists of designing a new calculus where only typed objects are formed. We will consider the typed version of the combinatorial completeness statement in both approaches.

In Curry's approach a new set of objects is defined, namely the types, and a new judgement form, that a combinator has a type. There are axioms and rules defining the provable judgments of this form. Thus, the notions and theorems involving the combinators are left as they were, and new theorems are stated involving the types. In the case of the combinatorial completeness, it is said that, if after assigning types to the variables of the etension an object can be assigned a type, then the abstraction of that object (defined for untyped combinators) can be assigned a type, namely the function type from the types of the variables to the type of the original object.

Following Church's approach, everything must be done again from the very begining. Once the types are defined, for each type it is said which are the combinators of that type; hence, instead of a set of combinators there is a family of sets, indexed by the types. The equality judgement is then stated for typed combinators. The extensions considered in order to define abstraction are extensions with typed variables and the abstraction is defined among typed combinators.

What follows presents the formalization of the definitions of abstraction and the proofs that it behaves as intended for the simply typed combinators following both approaches. Church's version is presented first in full detail, while Curry's approach is only sketched. Both were checked in Alf and the appendix contains the Alf implementation for Church's version.

3.1 Types a la Church

The types are formed from arbitrary atomic types by means of the function arrow $\rightarrow$. The objects are not introduced as a single set but as a family indexed by the types so there is a set of objects for each type. The elementary statements are equalities between objects of the same type. The restricted version of combinatorial completeness that can be proved takes the types of the incomplete object and the variable to be abstracted into account and forms a function of the adequate type. In order to describe the incomplete objects we must again consider extensions of the original theory by adjoining indeterminates, in this case typed indeterminates.

In the following definition of the theory of typed combinators, let α, β, γ, ... be variables of the epi-theory ranging over types. The objects of each type are defined from typed versions of K, S and application: For all types α, β, there is a constant

$$K_{\alpha,\beta} : \alpha \rightarrow \beta \rightarrow \alpha$$

For all types α, β, γ there is a constant

$$S_{\alpha,\beta,\gamma} : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$$

And for all types α, β there is an operation of application

$$\frac{M : \alpha \rightarrow \beta \quad N : \alpha}{(MN) : \beta}$$

The formulae are equalities between objects belonging to the same type, so if f and g are objects in the same type α, then $f \sim_\alpha g$ is an elementary formula. The theorems are characterized by essentially the same rules as in the untyped theory but also considering the typing of the objects involved:

$$K_{\alpha,\beta}ab \sim_\alpha a \qquad (K)$$

$$S_{\alpha,\beta,\gamma}fga \sim_\gamma fa(ga) \qquad (S)$$

$$f \sim_\alpha f \qquad (\rho)$$

$$\frac{f \sim_\alpha g}{g \sim_\alpha f} \qquad (\sigma)$$

$$\frac{f \sim_\alpha g \quad g \sim_\alpha h}{f \sim_\alpha h} \qquad (\tau)$$

$$\frac{f \sim_{\alpha\to\beta} g}{fa \sim_\beta ga} \qquad (\mu)$$

$$\frac{a \sim_\alpha b}{fa \sim_\beta fb} \qquad (\nu)$$

As an example of the objects definable in this theory, consider $S_{\alpha,\alpha\to\alpha,\alpha}K_{\alpha,\alpha\to\alpha}K_{\alpha,\alpha} : \alpha \to \alpha$, the typed identity function: let x be any object of type α, then

$$S_{\alpha,\alpha\to\alpha,\alpha}K_{\alpha,\alpha\to\alpha}K_{\alpha,\alpha}x \sim_\alpha K_{\alpha,\alpha\to\alpha}x(K_{\alpha,\alpha}x) \sim_\alpha x$$

We will have to consider extensions of this theory by adjoining indeterminates to express the intuitive incomplete objects. In order for an indeterminate to be adjoined, it must be introduced as an object of some type, and thus must have a type associated to it. We will use the notation x^α for an indeterminate x which has been assigned type α. In an extension with the indeterminates $x_1^{\alpha_1}, \ldots, x_n^{\alpha_n}$ the objects of a given type are built as above with the addition of letting variables of type α be objects of type α. An extension is then no longer a finite set, but a finite set together with an assignment of types to each of the elements of this set.

The formalization in set theory proceeds by considering that each theory consists of a family (not only a set as in the untyped case) of objects, formulae and theorems, and by considering a family of such theories indexed by extensions. Again the notion of extension must be made precise, in this case to take into account that there is no longer a set of objects to which indeterminates are adjoined, but a family of them. An extension of the theory of typed combinators may be characterized by a set of indeterminates together with a way of adjoining them to some sets of objects, thas is, together with a way of assigning types to them. Thus, not only the sets of indeterminates must be formalized (by the initial segments of the natural numbers) but also the type assignments to these sets. These will be formalized by finite sequences of types of length n for the type assignements for the initial segment of length n. The formalization, which is roughly as for the untyped case, requires the definition of the following sets:

- A set Type for the types of the combinators

  Type-formation

$$\mathsf{Type}\ Set$$

The set Type is inductively generated and there are two introduction rules; one for the atomic types, that is, the type variables, and one for function types. We let the type variables be indexed by some given set Atom:

Type-introduction 1

$$\frac{\alpha \in \mathsf{Atom}}{\mathsf{var}(\alpha) \in \mathsf{Type}}$$

Type-introduction 2

$$\frac{\alpha \in \mathsf{Type} \quad \beta \in \mathsf{Type}}{\alpha \to \beta \in \mathsf{Type}}$$

Type-elimination

$$\frac{\begin{array}{l} P(t)\ Set\ [t \in \mathsf{Type}] \\ b(at) \in P(\mathsf{var}(at))[at \in \mathsf{Atom}] \\ i(\alpha, \beta, I_\alpha, I_\beta) \in P(\alpha \to \beta)[\alpha, \beta \in \mathsf{Type}, I_\alpha \in P(\alpha), I_\beta \in P(\beta)] \\ \alpha \in \mathsf{Type} \end{array}}{\mathsf{TypeRec}(P, b, i, \alpha) \in P(t)}$$

$$\mathsf{TypeRec}(P, b, i, \mathsf{var}(at)) = b(at)$$
$$\mathsf{TypeRec}(P, b, i, \alpha \to \beta) = i(\alpha, \beta, \mathsf{TypeRec}(P, b, i, \alpha), \mathsf{TypeRec}(P, b, i, \beta))$$

- $\mathsf{IS}(\mathrm{n})$ *Set* $[\mathrm{n} \in \mathsf{N}]$
  The finite sets (the sets of indeterminates). The same sets as for the untyped case.

- A family TA for the type assignements to the sets of indeterminates:

  TA-Formation

  $$\mathsf{TA}(n)\ Set\ [n \in \mathsf{N}]$$

  Each element in $\mathsf{TA}(n)$ formalizes a type assignement for $\mathsf{IS}(n)$ as a sequence of length n of elements of Type:

  TA-introduction-1

  $$\mathsf{Nil} \in \mathsf{TA}(0)$$

  TA-introduction-2

  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha \in \mathsf{Type}}{\mathsf{Cons}(n, ta, \alpha) \in \mathsf{TA}(\mathsf{s}(n))}$$

  TA-elimination

  $$\frac{\begin{array}{l} P(x, y)\ Set\ [x \in \mathsf{N}, y \in \mathsf{TA}(x)] \\ e \in P(0, \mathsf{Nil}) \\ d(m, t, \alpha, i) \in P(\mathsf{s}(m), \mathsf{Cons}(m, t, \alpha))[m \in \mathsf{N}, t \in \mathsf{TA}(m), \alpha \in \mathsf{Type}, i \in P(m, t)] \\ n \in \mathsf{N} \\ ta \in \mathsf{TA}(n) \end{array}}{\mathsf{TAE}(P, e, d, n, ta) \in P(n, ta)}$$

$$\mathsf{TAE}(P,e,d,0,\mathsf{Nil}) = e$$
$$\mathsf{TAE}(P,e,d,\mathsf{s}(n),\mathsf{Cons}(n,t,\alpha)) = d(n,t,\alpha,\mathsf{TAE}(P,e,d,n,t))$$

- The family of objects of each type in each extension is defined by means of Ob. The set $\mathsf{Ob}(n,ta,\alpha)$ defines the set of objects of type α in an extension with n variables and a type assignement ta to the indeterminates of the extension. The introduction rules correspond to the formation of the terms:

  Ob-formation
  $$\mathsf{Ob}(n,ta,\alpha)\ \mathit{Set}\ [n \in \mathsf{N}, ta \in \mathsf{TA}(n), \alpha \in \mathsf{Type}]$$

  Ob-introduction-1
  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha,\beta \in \mathsf{Type}}{\mathsf{K}(n,ta,\alpha,\beta) \in \mathsf{Ob}(n,ta,\alpha \to \beta \to \alpha)}$$

  Ob-introduction-2
  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha,\beta,\gamma \in \mathsf{Type}}{\mathsf{S}(n,ta,\alpha,\beta,\gamma) \in \mathsf{Ob}(n,ta,(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma))}$$

  Ob-introduction-3
  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha \in \mathsf{Type} \quad x \in \mathsf{IS}(n) \quad p \in \mathsf{VarType}(n,ta,x,\alpha)}{\mathsf{Var}(n,ta,\alpha,x,p) \in \mathsf{Ob}(n,ta,\alpha)}$$

  In this last rule, which corresponds to introducing an indeterminate as an object of type α when the extension is given by n and ta, the premise $p \in \mathsf{VarType}(n,ta,x,\alpha)$ corresponds to a proof that the indeterminate x (an element in $\mathsf{IS}(n)$) has the type α in ta. The set $\mathsf{VarType}$, which formalizes this statement, is defined bellow.

  Ob-introduction-4
  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha,\beta \in \mathsf{Type} \quad f \in \mathsf{Ob}(n,ta,\alpha \to \beta) \quad a \in \mathsf{Ob}(n,ta,\alpha)}{\mathsf{App}(n,ta,\alpha,\beta,f,a) \in \mathsf{Ob}(n,ta,\beta)}$$

  Ob-elimination
  $$\frac{\begin{array}{l} n \in \mathsf{N} \\ ta \in \mathsf{TA}(n) \\ P(x,y)\ \mathit{Set}\ [x \in \mathsf{Type}, y \in \mathsf{Ob}(n,ta,x)] \\ k(u,v) \in P(u \to v \to u, \mathsf{K}(n,ta,u,v))\ [u,v \in \mathsf{Type}] \\ s(u,v,w) \in P((u \to v \to w) \to (u \to v) \to u \to w, \mathsf{S}(n,ta,u,v,w))\ [u,v,w \in \mathsf{Type}] \\ v(u,v,w) \in P(u, \mathsf{Var}(n,ta,u,v,w))\ [u \in \mathsf{Type}, v \in \mathsf{IS}(n), w \in \mathsf{VarType}(n,ta,v,u)] \\ a(u,v,w,x,iw,ix) \in P(v, \mathsf{App}(n,ta,u,v,w,x)) \left[\begin{array}{l} u,v \in \mathsf{Type}, \\ w \in \mathsf{Ob}(n,ta,u \to v), \\ x \in \mathsf{Ob}(n,ta,u), \\ iw \in P(u \to v, w), \\ ix \in P(u,x) \end{array}\right] \\ \alpha \in \mathsf{Type} \\ p \in \mathsf{Ob}(n,ta,\alpha) \end{array}}{\mathsf{obE}(n,ta,P,k,s,v,a,\alpha,t) \in P(\alpha,t)}$$

$$
\begin{aligned}
&\mathsf{obE}(n, ta, P, k, s, v, a, \alpha \to \beta \to \alpha, \mathsf{K}(n, ta, \alpha, \beta)) = k(\alpha, \beta)\\
&\mathsf{obE}(n, ta, P, k, s, v, a, (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathsf{S}(n, ta, \alpha, \beta, \gamma))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = s(\alpha, \beta, \gamma)\\
&\mathsf{obE}(n, ta, P, k, s, v, a, \alpha, \mathsf{Var}(n, ta, \alpha, x, p)) = v(\alpha, x, p)\\
&\mathsf{obE}(n, ta, P, k, s, v, a, \beta, \mathsf{App}(n, ta, \alpha, \beta, f, x)) = a(\alpha, \beta, f, x,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{obE}(n, ta, P, k, s, v, a, \alpha \to \beta, f),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{obE}(n, ta, P, k, s, v, a, \alpha, x))
\end{aligned}
$$

- In order to formalize the variable x^α we will define a set of proofs of a variable (an element in some IS) being assigned a type by a type assignement, i.e. a set that formalizes that the type of the i-th element in an initial segment is assigned the type in the i-th possition of the type assignment. $\mathsf{VarType}(n, ta, x, \alpha)$ defines the predicate stating that the type assigned to x by the type assignement ta is α.

  VarType-formation

  $$\mathsf{VarType}(n, ta, x, \alpha)\ \mathit{Set}\ [n \in \mathsf{N}, ta \in \mathsf{TA}, x \in \mathsf{IS}(n), \alpha \in \mathsf{Type}]$$

  VarType-introduction-1

  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha \in \mathsf{Type}}{\mathsf{PutType}(n, ta, \alpha) \in \mathsf{VarType}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \mathsf{put}(n), \alpha)}$$

  VarType-introduction-2

  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha, \beta \in \mathsf{Type} \quad y \in \mathsf{IS}(n) \quad p \in \mathsf{VarType}(n, ta, \alpha, y)}{\mathsf{InjectType}(n, ta, \alpha, \beta, y, p) \in \mathsf{VarType}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \beta), \mathsf{inject}(n, y), \alpha)}$$

  The first introduction rule states that the type assigned to the last variable in an initial segment of length n is the last type in a type assignement of this length. The second one states that the type of a variable present already in the previous initial segment has the type given to it in the corresponding shorter type assignment.

  VarType-elimination

  $$\frac{\begin{array}{l}
n \in \mathsf{N}\\
ta \in \mathsf{TA}(n)\\
P(x, y, z, t)\ \mathit{Set}\left[\begin{array}{l} x \in \mathsf{IS}(\mathsf{s}(n)),\\ y, z \in \mathsf{Type},\\ t \in \mathsf{VarType}(\mathsf{s}(n), \mathsf{Cons}(n, ta, z), x, y)\end{array}\right]\\
e(x) \in P(\mathsf{put}(n), x, x, \mathsf{PutType}(n, ta, x))\ [x \in \mathsf{Type}]\\
d(x, y, z, t) \in P(\mathsf{inject}(n, z), x, y, \mathsf{InjectType}(n, ta, x, y, z, t))\left[\begin{array}{l} x, y \in \mathsf{Type},\\ z \in \mathsf{IS}(n),\\ t \in \mathsf{VarType}(n, ta, z, x)\end{array}\right]\\
i \in \mathsf{IS}(\mathsf{s}(n))\\
\alpha, \beta \in \mathsf{Type}\\
p \in \mathsf{VarType}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \beta), i, \alpha)
\end{array}}{\mathsf{VarTypeE}(n, ta, P, e, d, i, \alpha, \beta, p) \in P(i, \alpha, \beta, p)}$$

$$\mathsf{VarTypeE}(n, ta, P, e, d, \mathsf{put}, \alpha, \alpha, \mathsf{PutType}(n, ta, \alpha)) = e(\alpha)$$
$$\mathsf{VarTypeE}(n, ta, P, e, d, \mathsf{inject}(n, x), \alpha, \beta, \mathsf{InjectType}(n, ta, \alpha, \beta, x, p)) = d(\alpha, \beta, x, p)$$

- There is a set which formalizes the formulae of the typed combinators, with only one introduction rule, corresponding to the only way of building formulae, i.e. equality between objects of the same type in a given extension:

  Form-formation

  $$\mathsf{Form}(n, ta, \alpha)\ \mathit{Set}\ [n \in \mathsf{N}, ta \in \mathsf{TA}(n), \alpha \in \mathsf{Type}]$$

  Form-introduction

  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha \in \mathsf{Type} \quad f, g \in \mathsf{Ob}(n, ta, \alpha)}{f \sim_{n,ta} g \in \mathsf{Form}(n, ta, \alpha)}$$

- Finally, there is a set describing the theorems, i.e. the provable equalities. The introduction rules correspond to the axioms and inference rules. As in the untyped case we shall only present some of them, the ones corresponding to (K), (ρ) and (μ).

  $\vdash$-formation

  $$\vdash_{n,ta} f\ \mathit{Set}\ [n \in \mathsf{N}, ta \in \mathsf{TA}(n), \alpha \in \mathsf{Type}, f \in \mathsf{Form}(n, ta, \alpha)]$$

  $\vdash$-introduction-K

  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha, \beta \in \mathsf{Type} \quad a \in \mathsf{Ob}(n, ta, \alpha), b \in \mathsf{Ob}(n, ta, \beta)}{\mathsf{kk}(n, ta, \alpha, \beta, a, b) \in \vdash_{n,ta} \mathsf{App}(\mathsf{App}(n, ta, \alpha, \beta \to \alpha, \mathsf{K}, a), n, ta, \beta, alpha, b) \sim_{n,ta} a}$$

  $\vdash$-introduction-ρ

  $$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha \in \mathsf{Type} \quad a \in \mathsf{Ob}(n, ta, \alpha)}{\rho(n, ta, \alpha, a) \in \vdash_{n,ta} a \sim_{n,ta} a}$$

  $\vdash$-introduction-μ

  $$\frac{\begin{array}{l} n \in \mathsf{N} \\ ta \in \mathsf{TA}(n) \\ \alpha, \beta \in \mathsf{Type} \\ f, g \in \mathsf{Ob}(n, ta, \alpha \to \beta) \\ a \in \mathsf{Ob}(n, ta, \alpha) \\ p \in \vdash_{n,ta} f \sim_{n,ta} g \end{array}}{\mu(n, ta, \alpha, \beta, f, g, a, p) \in \vdash_{n,ta} \mathsf{App}(n, ta, \alpha, \beta, f, a) \sim_{n,ta} \mathsf{App}(n, ta, \alpha, \beta, g, a)}$$

As for the untyped case, there is a natural way of thinking of an object in a theory as an object in an extension, as built up in the same way. This is made explicit by a function $\mathcal{O}$ (as before) which takes an object in a theory and builds up the corresponding one in the extension. For this typed theory, we will only consider the abstraction of one variable, the induction on n required to abstract all the n variables of an extension is essentially the same as for the untyped theory. Thus we will only require the injection of a theory into the next one, which is provided by the function

$$\mathcal{O} \in (\Pi n \in \mathsf{N})(\Pi ta \in \mathsf{TA}(n))(\Pi \alpha, \beta \in \mathsf{Type})(\mathsf{Ob}(n, ta, \beta)) \supset \mathsf{Ob}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \beta)$$

The function $\mathcal{O}$ takes a natural number n, a type assignment ta of length n, a type α to extend the type assignment and a type β which is the type of the object in question. It then takes an object of type β in an extension with n indeterminates which are assigned types by ta and produces an object of type β built in the same way in a theory with $\mathsf{s}(n)$ indeterminates assigned types by $\mathsf{Cons}(n, ta, \alpha)$. The definition is by induction on $\mathsf{Ob}(n, ta, \beta)$, formally by application of the elimination rule for this set.

This concludes the formalization of the theory of simply typed combinators, the next section considers the definition of the typed version of abstraction.

3.1.1 Typed Abstraction

Consider an extension with n indeterminates $x_1^{\alpha_1}, \ldots, x_n^{\alpha_n}$. By adding one more indeterminate x^α, the objects $h' : \beta$ in this new theory are such that when replacing an object of type α for x^α in them, they yield an object of type β. We will show that these objects were already present in the theory without x^α as functions $h : \alpha \to \beta$. This will be done by defining for each $h' : \beta$ an object $[x^\alpha]h' : \alpha \to \beta$ and then showing that

$$([x^\alpha]h')x^\alpha \sim_{\mathsf{s}(n),\beta} h'$$

Both the definition of $[x^\alpha]h'$ and the proof of $([x^\alpha]h')x^\alpha \sim_{\mathsf{s}(n),\beta} h'$ are by induction on h' being an object of type β in the extension with $x_1^{\alpha_1}, \ldots, x_n^{\alpha_n}, x^\alpha$. Except for the type information involved, the definition proceeds in the same way as for the untyped case in all cases but the one for the variable. In the untyped case, the analysis of which variable is considered is by induction on the definition of being in the corresponding initial segment, which alone characterizes the extension. In the typed case, the analysis is an induction on the definition of which type is assigned to the variable in the extension, because it is this definition which follows the construction of the extension, it takes into account both the initial segment and the type assignement. The definition of $[x^\alpha]h'$ will be done by analyzing what function of x^α h' is for each of the forms of constructing h':

- $h' : \beta$ is K_{γ_1,γ_2} (β is $\gamma_1 \to \gamma_2 \to \gamma_1$). When thought of as a function of x^α it is the constant function with value K_{γ_1,γ_2}. Thus we define $[x^\alpha]K_{\gamma_1,\gamma_2} : \alpha \to (\gamma_1 \to \gamma_2 \to \gamma_1)$ to be $K_{\gamma_1\to\gamma_2\to\gamma_1,\alpha}K_{\gamma_1,\gamma_2}$.
- $h' : \beta$ is $S_{\gamma_1,\gamma_2,\gamma_3}$ (β is $(\gamma_1 \to \gamma_2 \to \gamma_3) \to (\gamma_1 \to \gamma_2) \to \gamma_1 \to \gamma_3$). With the same argument as for K_{γ_1,γ_2} we define $[x^\alpha]S_{\gamma_1,\gamma_2,\gamma_3} : \alpha \to ((\gamma_1 \to \gamma_2 \to \gamma_3) \to (\gamma_1 \to \gamma_2) \to \gamma_1 \to \gamma_3)$ to be $K_{(\gamma_1\to\gamma_2\to\gamma_3)\to(\gamma_1\to\gamma_2)\to\gamma_1\to\gamma_3,\alpha}S_{\gamma_1,\gamma_2,\gamma_3}$.

- $h' : \beta$ is a variable of the extension. Then there are two cases to consider according to whether the variable is x^α (β is α) or some other variable y^γ (β is γ). In the case it is y^γ, by the same argument as for K_{γ_1,γ_2} and $S_{\gamma_1,\gamma_2,\gamma_3}$, the definition of $[x^\alpha]y^\gamma : \alpha \to \gamma$ is $K_{\gamma,\alpha}y^\gamma$. In the case it is x^α the definition is such that the function is the identity function, as x^α is when considered a function of x^α. Thus, for this case we define $[x^\alpha]x^\alpha : \alpha \to \alpha$ to be $S_{\alpha,\alpha\to\alpha,\alpha}K_{\alpha,\alpha\to\alpha}K_{\alpha,\alpha}$ which was shown above to behave as the identity.

- $h' : \beta$ is an application fg where both $f : \gamma_1 \to \gamma_2$ and $g : \gamma_1$ may contain x^α and thus are to be considered as functions of x^α. The induction hypothesis yields the definitions of the two functions $[x^\alpha]f : \alpha \to \gamma_1 \to \gamma_2$ and $[x^\alpha]g : \alpha \to \gamma_1$. The object that results when performing the substitution of an object of type α for x^α in fg may be analyzed as formed by the application of the object that results from performing the substitution in f to the one that results from performing it in g. In terms of $[x^\alpha]f$ and $[x^\alpha]g$ this is expressed by the application of $([x^\alpha]f)o$ to $([x^\alpha]g)o$, where o is the object substituted for x^α. We thus define $[x^\alpha](fg) : \alpha \to \gamma_2$ to be $S_{\alpha\to\gamma_1\to\gamma_2,\alpha\to\gamma_1,\alpha}\,([x^\alpha]f)([x^\alpha]g)$.

The formalization of this definition is as follows. Given $n \in \mathsf{N}$, $ta \in \mathsf{TA}(n)$, $\alpha \in \mathsf{Type}$ and $h' \in \mathsf{Ob}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \beta)$, i.e. an object of type β in an extension with $\mathsf{s}(n)$ indeterminates, the last of which is assigned type α; apply the elimination rule for Ob to h' to find an element in $\mathsf{Ob}(n, ta, \alpha \to \beta)$ according to the definition above. The premises of this rule yield the following cases:

- h' is $\mathsf{K}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \gamma_1, \gamma_2)$. Then define the abstraction to be the constant function with value K in $\mathsf{Ob}(n, ta, \alpha \to (\gamma_1 \to \gamma_2 \to \gamma_1)$:

$$\mathsf{App}(n, ta, \mathsf{K}(n, ta, \gamma_1 \to \gamma_2 \to \gamma_1, \alpha), \mathsf{K}(n, ta, \gamma_1, \gamma_2))$$

(note that we did not write the type parameters of App. They may be restored by looking at the types of the arguments. We will continue with this convention along this definition).

- h' is $\mathsf{S}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \gamma_1, \gamma_2, \gamma_3)$. Then define the abstraction to be the constant function with value S in $\mathsf{Ob}(n, ta, \alpha \to (\gamma_1 \to \gamma_2 \to \gamma_3) \to (\gamma_1 \to \gamma_2) \to \gamma_1 \to \gamma_3)$:

$$\mathsf{App}(n, ta, \mathsf{K}(n, ta, (\gamma_1 \to \gamma_2 \to \gamma_3) \to (\gamma_1 \to \gamma_2) \to \gamma_1 \to \gamma_3, \alpha), \mathsf{S}(n, ta, \gamma_1, \gamma_2, \gamma_3))$$

- h' is $\mathsf{Var}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \gamma, x, p)$, where $x \in \mathsf{IS}(\mathsf{s}(n))$, and

$$p \in \mathsf{VarType}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), x, \gamma)$$

is a proof that the type assigned to x in the extension is γ. Then one must analyze whether x is the variable to be abstracted (the variable that extends the theory from n to $\mathsf{s}(n)$) or any other variable (already present in the theory for n). This is done by making an induction on p, i.e. by applying the elimination rule for

$$\mathsf{VarType}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), x, \gamma)$$

on p. The two cases that arise correspond to whether x is the last variable of the extension, the one we are abstracting, or some other:

– p is $\mathsf{PutType}(n, ta, \alpha)$. Then define the abstraction to be the identity function in $\mathsf{Ob}(n, ta, \alpha \to \alpha)$:

$$\mathsf{App}(n, ta, \mathsf{App}(n, ta, \mathsf{S}(n, ta, \alpha, \alpha \to \alpha, \alpha), \mathsf{K}(n, ta, \alpha, \alpha \to \alpha))\mathsf{K}(n, ta, \alpha, \alpha))$$

– p is $\mathsf{InjectType}(n, ta, \delta, \alpha, y, q)$ where y is in $\mathsf{IS}(n)$ (a variable already present in the extension with only n variables) and $q \in \mathsf{VarType}(n, ta, y, \delta)$ is a proof that the type assigned to y in this theory is δ. Then define the abstraction to be the constant function with value y in $\mathsf{Ob}(n, ta, \alpha \to \delta)$:

$$\mathsf{App}(n, ta, \mathsf{K}(n, ta, \delta, \alpha), \mathsf{Var}(n, ta, \delta, y, q)).$$

- h' is an application $\mathsf{App}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \gamma_1, \gamma_2, f, a)$, where

$$f \in \mathsf{Ob}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \gamma_1 \to \gamma_2)$$

$$a \in \mathsf{Ob}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \gamma_1)$$

In this case the elimination rule provides also induction hypothesis for f and a which for this definition yield that both the abstraction for f and a are defined, say $fh \in \mathsf{Ob}(n, ta, \alpha \to \gamma_1 \to \gamma_2)$ and $ah \in \mathsf{Ob}(n, ta, \alpha \to \gamma_1)$. Then the corresponding object for the application is

$$\mathsf{App}(n, ta, \mathsf{App}(n, ta, \mathsf{S}(n, ta, \alpha \to \gamma_1 \to \gamma_2, \alpha \to \gamma_1, \alpha)), fh), ah).$$

The next step is to prove that the defined function has the intended behaviour, i.e. that $([x^\alpha]h')x^\alpha \sim_{\mathsf{s}(n),\beta} h'$. This proof is again by induction on $h' : \beta$ and as for the untyped case follows the definition of $([x^\alpha]h')$. It is formalized by an application of the elimination rule for $\mathsf{Ob}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \beta)$ and also follows the definition of the function. From both the definition of the function and the proof about its behaviour we obtain a proof of

$$\begin{aligned}&(\Pi n \in \mathsf{N})(\Pi ta \in \mathsf{TA}(n))(\Pi \alpha, \beta \in \mathsf{Type})(\Pi h' \in \mathsf{Ob}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), \beta))\\ &\quad (\Sigma h \in \mathsf{Ob}(n, ta, \alpha \to \beta))\mathsf{App}(\mathcal{O}(h), \mathsf{put}(n, ta, \mathsf{PutType}(n, ta, \alpha))) \sim_{\mathsf{s}(n),\beta} h'\end{aligned}$$

where $\mathsf{put}(n, ta, \mathsf{PutType}(n, ta, \alpha))$ is the variable added with type α.

The formalization of this proof, as it was edited and checked with the Alf system may be found in the appendix.

3.2 Types a la Curry

This approach is based on analysing which (untyped) combinators may be assigned a type. Consider the theory of untyped combinators and define a set for the types as in the previous section. A new judgement form is introduced, $M : \alpha$ where M is an (untyped) combinator and α is a type. The axioms and rules defining the provable judgments of this form are:

For any types α, β, it is provable that

$$K : \alpha \to \beta \to \alpha$$

For any types α, β, γ, it is provable that

$$S : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

For any types α, β,

$$\frac{M : \alpha \to \beta \quad N : \alpha}{MN : \beta}$$

As opposed to the system a la Church where there is a different constant $K_{\alpha,\beta}$ for each different types α, β, here there is only one constant K which may be assigned many types.

The extensions and the definition of abstraction are the ones for the untyped combinators. It remains to be said when a combinator in an extension with indeterminates $x_1, \ldots, x_n$may be assigned a type. To do this, we consider again type asignments to the indeterminates and extend the notion of having a type to having a type in an extension given a type assignment. The formalization proceeds by defining the set TypeOf:

TypeOf-formation

$$\mathsf{TypeOf}(n, ta, M, \alpha) \ \mathit{Set} \ [n \in \mathsf{N}, ta \in \mathsf{TA}(n), M \in \mathsf{Ob}(n), \alpha \in \mathsf{Type}$$

TypeOf-introduction-1

$$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha, \beta \in \mathsf{Type}}{\mathsf{Kt}(n, ta, \alpha, \beta) \in \mathsf{TypeOf}(n, ta, \mathsf{K}, \alpha \to \beta \to \alpha)}$$

TypeOf-introduction-2

$$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad \alpha, \beta, \gamma \in \mathsf{Type}}{\mathsf{St}(n, ta, \alpha, \beta, \gamma) \in \mathsf{TypeOf}(n, ta, \mathsf{S}, (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma)}$$

TypeOf-introduction-3

$$\frac{n \in \mathsf{N} \quad ta \in \mathsf{TA}(n) \quad x \in \mathsf{IS}(n) \quad \alpha \in \mathsf{Type} \quad p \in \mathsf{VarType}(n, ta, x, \alpha)}{\mathsf{vart}(n, ta, x, \alpha, p) \in \mathsf{TypeOf}(n, ta, \mathsf{var}(n, x), \alpha)}$$

TypeOf-introduction-4

$$\frac{\begin{array}{l} n \in \mathsf{N} \\ ta \in \mathsf{TA}(n) \\ f, a \in \mathsf{Ob}(n) \\ \alpha, \beta \in \mathsf{Type} \\ p \in \mathsf{TypeOf}(n, ta, f, \alpha \to \beta) \\ q \in \mathsf{TypeOf}(n, ta, a, \alpha) \end{array}}{\mathsf{appt}(n, ta, \alpha, \beta, f, a, p, q) \in \mathsf{TypeOf}(n, ta, \mathsf{App}(n, f, a), \beta)}$$

That abstraction behaved as intended was already shown for the untyped combinators; it is shown now that it also behaves adequately with respect to the types. We show that if M is a combinator in an extension with indeterminates $x_1, \ldots, x_n$ then, if given a type assignment $\alpha_1, \ldots, \alpha_n$ it is the case that $M : \beta$, then it is the case that $[x_1, \ldots, x_n]M : \alpha_1 \to \ldots \alpha_n \to \beta$. This is proved by induction on n based on the proof that if M is a combinator in an extension with indeterminates $x_1, \ldots, x_n, x$ which can be assigned type β under the type assignment $\alpha_1, \ldots, \alpha_n, \alpha$, then $[x]M$ is assigned type $\alpha \to \beta$ in the extension $x_1, \ldots, x_n$ under the type assignment $\alpha_1, \ldots, \alpha_n$. This, in turn, is proved by induction on $\mathsf{TypeOf}(\mathsf{s}(n), \mathsf{Cons}(n, ta, \alpha), M, \beta)$. This theorem stating the well behaviour of abstraction with respect to types is a specialization of Curry's stratification theorem as presented in [CF58].

3.3 Remarks

The typed combinators constitute the Curry-Howard interpretation of minimal logic as presented in the Hilbert style. The types are interpreted as the propositions (the arrow corresponds to implication) and the terms as codes for the proofs in the system. Thus the constants $K_{\alpha,\beta}$ and $S_{\alpha,\beta,\gamma}$ code the proofs of the axioms given by their types; and application codes a proof formed using modus monens. The indeterminates in an extension should then be interpreted as proofs of assumptions which are given by the type assigned to them. The theorem about the restricted combinatorial completeness can be read under this interpretation as stating the deductive completeness of the logical system. It states that if there is a proof of proposition β under the assumption of proposition α then there is a proof of the proposition $\alpha \supset \beta$. The argument above for the typed combinators may be seen as a proof of this statement by providing a proof for $\alpha \supset \beta$ given the hypothetical proof of β.

There is a much simpler way (suggested by several people) to formalize the typed extensions when following Church's approach. An extension is charachterized by a list of types; then, a variable of a given type in such an extension is identified with the proof that that type is in a given position in the list. This approach was also verified in Alf. The formalization is based in changing all definitions so that the parameter charachterizing the extension is only one, a list of types.

Acknowledgments

I am most grateful to Bengt Nordström and Jan Smith for their guidance, encouragement and good disposition. I also owe thanks to Daniel Fridlender and Nora Szasz for many helpful ideas and the careful reading of this work, and to the Programming Methodology Group for providing a very stimulating environment.

References

[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, 1958.

[Fre84] G. Frege. Function and Concept. In B. McGuinness, editor, *Gottlob Frege, Collected Papers on Mathematics, Logic, and Philosophy*, pages 137–156. Basil Blackwell, 1984.

[Kle52] S. C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1952.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[San67] L. E. Sanchís. Functionals defined by Recursion. *Notre Dame Journal of Formal Logic*, 8, 1967.

[Sch24] M. Schönfinkel. Über die Bausteine der Mathematischen Logik. *Mathematische Annalen*, 92:305, 1924.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

Appendix

This appendix contains the result of editing in Alf the formalizations of the untyped and typed combinators and the proof of combinatorial completeness for them. In Alf there is a basic type of sets Set and it is possible to form function types by means of the construction $(x : A)B$ where B may depend on x. The elements of such types are formed by abstraction, $[x]b$ is a term in $(x : A)B$ for $b : B$ under the assumption that $x : A$. These elements may be used by application $f(a)$.

The implementation in Alf consists of declaring a new constant of type Set for every set former, or of a function type with values in Set for every family former. Then, corresponding to each introduction and elimination rule, a constant is declared with its type corresponding to the type of functions from the types corresponding premisses to the type corresponding to the conclusion. The definitional equalities are introduced as equalities in Alf.

In Martin-Löf's set theory there is the type of sets and then for every set there is type of elements of that set. Alf provides an implementation of this by letting every constant of type Set to be a type.

The code for the proof in Alf for both the untyped and typed combinators begins with the code for the basic sets of Martin-Löf's set theory.

Basic Sets

The set $\perp$ for absurdity

$$\perp : \mathsf{Set}$$

$$\perp_e : (P : (x : \perp)\mathsf{Set})(bott : \perp)P(bott)$$

The set of natural numbers

$$\mathsf{N} : \mathsf{Set}$$

$$\mathsf{0} : \mathsf{N}$$

$$\mathsf{s} : (\mathsf{N})\mathsf{N}$$

$$\mathsf{natrec} : (P : (\mathsf{N})\mathsf{Set})(b : P(\mathsf{0}))(i : (x : \mathsf{N})(P(x))P(\mathsf{s}(x)))(n : \mathsf{N})P(n)$$

$$\mathsf{natrec}(P, b, i, \mathsf{0}) = b$$

$$\mathsf{natrec}(P, b, i, \mathsf{s}(n)) = i(n, \mathsf{natrec}(P, b, i, n))$$

The set $<$ for the order relation between natural numbers

$$<: (\mathsf{N})(\mathsf{N})\mathsf{Set}$$

$$\mathsf{OneStep} : (n : \mathsf{N})\, n < \mathsf{s}(n)$$

$$\mathsf{MoreSteps} : (m : \mathsf{N})(n : \mathsf{N})(o : \mathsf{N})(m < n\ )(n < o\ )\, m < o$$

$$
\begin{aligned}
<_e \quad : \quad & (P : (m : \mathsf{N})(n : \mathsf{N})(l : m < n\ )\mathsf{Set}) \\
& (e : (n : \mathsf{N})P(n, \mathsf{s}(n), \mathsf{OneStep}(n))) \\
& (d \quad : \quad (m : \mathsf{N})(n : \mathsf{N})(o : \mathsf{N}) \\
& \qquad\quad (lmn : m < n\ ) \\
& \qquad\quad (lno : n < o\ ) \\
& \qquad\quad (imn : P(m, n, lmn)) \\
& \qquad\quad (ino : P(n, o, lno)) \\
& \qquad\quad P(m, o, \mathsf{MoreSteps}(m, n, o, lmn, lno))) \\
& (x : \mathsf{N}) \\
& (y : \mathsf{N}) \\
& (lxy : x < y\ ) \\
& P(x, y, lxy)
\end{aligned}
$$

$$<_e(P, e, d, x, \mathsf{s}(x), \mathsf{OneStep}(x)) = e(x)$$

$$
\begin{aligned}
& <_e(P, e, d, x, z, \mathsf{MoreSteps}(x, y, z, l_1, l_2)) = \\
& \qquad\qquad\qquad d(x, y, z, l_1, l_2, <_e(P, e, d, x, y, l_1), <_e(P, e, d, y, z, l_2))
\end{aligned}
$$

The set corresponding to disjunction

$$\vee : (\mathsf{Set})(\mathsf{Set})\mathsf{Set}$$

$$\mathsf{inl} : (A : \mathsf{Set})(B : \mathsf{Set})(a : A)A \vee B$$

$$\mathsf{inr} : (A : \mathsf{Set})(B : \mathsf{Set})(b : B)A \vee B$$

$$
\begin{aligned}
\vee_e \quad : \quad & (A : \mathsf{Set}) \\
& (B : \mathsf{Set}) \\
& (P : (A \vee B)\mathsf{Set}) \\
& (el : (a : A)P(\mathsf{inl}(A, B, a))) \\
& (er : (b : B)P(\mathsf{inr}(A, B, b))) \\
& (p : A \vee B) \\
& P(p)
\end{aligned}
$$

$$\vee_e(A, B, P, el, er, \mathsf{inl}(A, B, a)) = el(a)$$

$$\vee_e(A, B, P, el, er, \mathsf{inr}(A, B, b)) = er(b)$$

Note that $\vee$ is used as infix, $A \vee B$ is an abbreviation for $\vee(A, B)$. The same is done for the conjunction and the implication.

The set corresponding to the universal quantifier

$$\Pi : (A : \mathsf{Set})(B : (A)\mathsf{Set})\mathsf{Set}$$

$$\Lambda : (A : \mathsf{Set})(B : (A)\mathsf{Set})(f : (x : A)B(x))\Pi(A, B)$$

$$\Pi_e : (A : \mathsf{Set})(B : (A)\mathsf{Set})(f : \Pi(A, B))(a : A)B(a)$$

$$\Pi_e(A, B, \Lambda(A, B, f), a) = f(a)$$

The set of functions corresponding to implication

$\supset: (A : \mathsf{Set})(B : \mathsf{Set})\mathsf{Set}$

$\lambda : (A : \mathsf{Set})(B : \mathsf{Set})(f : (A)B)A \supset B$

$\supset_e : (A : \mathsf{Set})(B : \mathsf{Set})(f : A \supset B)(a : A)B$

$\supset_e(A, B, \lambda(A, B, f), a) = f(a)$

The set for propositional equality

$=: (A : \mathsf{Set}; a, b : A)\mathsf{Set}$

$\mathsf{id} : (A : \mathsf{Set}; x : A)x =_A x$

$\mathsf{J} : (A : \mathsf{Set}; C : (x : A)(y : A)(z : x =_A y)\mathsf{Set}; d : (x : A)C(x, x, \mathsf{id}(A, x)); a, b : A; p : a =_A b)C(a, b, p)$

$\mathsf{J}(A, C, d, a, a, \mathsf{id}(A, a)) = d(a)$

where we use $a =_A b$ as an abreviation for $= (A, a, b)$.

The initial segments of natural numbers

$\mathsf{IS} : (\mathsf{N})\mathsf{Set}$

$\mathsf{put} : (n : \mathsf{N})\mathsf{IS}(\mathsf{s}(n))$

$\mathsf{inject} : (n : \mathsf{N})(i : \mathsf{IS}(n))\mathsf{IS}(\mathsf{s}(n))$

$$
\begin{array}{lll}
\mathsf{IS}_e & : & (n : \mathsf{N}) \\
& & (P : (\mathsf{IS}(\mathsf{s}(n)))\mathsf{Set}) \\
& & (e : P(\mathsf{put}(n))) \\
& & (d : (i : \mathsf{IS}(n))P(\mathsf{inject}(n, i))) \\
& & (p : \mathsf{IS}(\mathsf{s}(n))) \\
& & P(p)
\end{array}
$$

$\mathsf{IS}_e(n, P, e, d, \mathsf{put}(n)) = e$

$\mathsf{IS}_e(n, P, e, d, \mathsf{inject}(n, i)) = d(i)$

The proof that an initial segment may be injected into one of greater length:

$\mathsf{ISInjection} : (m : \mathsf{N})(n : \mathsf{N})(p : m < n \ \vee m =_N n)\mathsf{IS}(m) \supset \mathsf{IS}(n)$

$$
\begin{aligned}
\mathsf{ISInjection}\ = [m,n,p]\vee_e(&m < n\ ,\\
&m =_N n,\\
&[p_1]\mathsf{IS}(m) \supset \mathsf{IS}(n),\\
&[a]{<_e}([m,n,a]IS(m) \supset IS(n),\\
&\qquad [n_1]\lambda(\mathsf{IS}(n_1),\mathsf{IS}(\mathsf{s}(n_1)),[h]\mathsf{inject}(n_1,h)),\\
&\qquad [m_1,n_1,o,lmn,lno,hlmn,hlno]\\
&\qquad\quad \lambda(\mathsf{IS}(m_1),\mathsf{IS}(o),\\
&\qquad\qquad [h_1]{\supset_e}(\mathsf{IS}(n_1),\\
&\qquad\qquad\qquad \mathsf{IS}(o),\\
&\qquad\qquad\qquad hlno,\\
&\qquad\qquad\qquad {\supset_e}(\mathsf{IS}(m_1),\mathsf{IS}(n_1),hlmn,h_1))),\\
&\qquad m,\\
&\qquad n,\\
&\qquad a),\\
&[b]\mathsf{J}(\mathsf{N},[x,y,p]\mathsf{IS}(x) \supset \mathsf{IS}(y),[x_1]\lambda(\mathsf{IS}(x_1),\mathsf{IS}(x_1),[h]h),m,n,b),\\
&p)
\end{aligned}
$$

Untyped Combinators

The objects

$\mathsf{Ob} : (n : \mathsf{N})\mathsf{Set}$

$\mathsf{K} : (n : \mathsf{N})\mathsf{Ob}(n)$

$\mathsf{S} : (n : \mathsf{N})\mathsf{Ob}(n)$

$\mathsf{var} : (n : \mathsf{N}; a : \mathsf{IS}(n))\mathsf{Ob}(n)$

$\mathsf{app} : (n : \mathsf{N}; f, a : \mathsf{Ob}(n))\mathsf{Ob}(n)$

$$
\begin{aligned}
\mathsf{Ob}_e : (&n : \mathsf{N})\\
&(P : (x : \mathsf{Ob}(n))\mathsf{Set})\\
&(e : P(\mathsf{K}(n)))\\
&(f : P(\mathsf{S}(n)))\\
&(g : (x : \mathsf{IS}(n))P(\mathsf{var}(n,x)))\\
&(h : (x : \mathsf{Ob}(n))(y : \mathsf{Ob}(n))(z : P(x))(t : P(y))P(\mathsf{app}(n,x,y)))\\
&(o : \mathsf{Ob}(n))\\
&P(o)
\end{aligned}
$$

$\mathsf{Ob}_e(n,P,e,f,g,h,\mathsf{K}(n)) = e$

$\mathsf{Ob}_e(n,P,e,f,g,h,\mathsf{S}(n)) = f$

$\mathsf{Ob}_e(n,P,e,f,g,h,\mathsf{var}(n,a)) = g(a)$

$\mathsf{Ob}_e(n,P,e,f,g,h,\mathsf{app}(n,a,b)) = h(a,b,\mathsf{Ob}_e(n,P,e,f,g,h,a),\mathsf{Ob}_e(n,P,e,f,g,h,b))$

The term that codes the Identity:

$\mathsf{I} = [n]\mathsf{app}(n,\mathsf{app}(n,\mathsf{S}(n),\mathsf{K}(n)),\mathsf{K}(n)) : (n : \mathsf{N})\mathsf{Ob}(n)$

The formulae

$$\mathsf{form} : (n : \mathsf{N})\mathsf{Set}$$

$$\sim: (n : N; f, g : \mathsf{Ob}(n))\mathsf{form}(n)$$

$$\begin{aligned}\mathsf{form}_e : &(n : N)\\ &(P : (\mathsf{form}(n))\mathsf{Set})\\ &(e : (x : \mathsf{Ob}(n))(y : \mathsf{Ob}(n))P(x \sim_n y))\\ &(f : \mathsf{form}(n))\\ &P(f)\end{aligned}$$

$$\mathsf{form}_e(n, P, e, f \sim_n g) = e(f, g)$$

Where we use $a \sim_n b$ as an abreviation for $\sim (n, a, b)$.

The theorems

$$\vdash: (n : \mathsf{N})(f : \mathsf{form}(n))\mathsf{Set}$$

$$\mathsf{Keq} : (n : \mathsf{N})(M, P : \mathsf{Ob}(n)) \vdash_n \mathsf{app}(n, \mathsf{app}(n, \mathsf{K}(n), M), P) \sim_n M$$

$$\begin{aligned}\mathsf{Seq} : &(n : \mathsf{N})(M, P, O : \mathsf{Ob}(n))\\ &\vdash_n \mathsf{app}(n, \mathsf{app}(n, \mathsf{app}(n, \mathsf{S}(n), M), P), O) \sim_n \mathsf{app}(n, \mathsf{app}(n, M, O), \mathsf{app}(n, P, O))\end{aligned}$$

$$\mu : (n : \mathsf{N})(M, O, P : \mathsf{Ob}(n))(p : \vdash_n M \sim_n O) \vdash_n \mathsf{app}(n, M, P) \sim_n \mathsf{app}(n, O, P)$$

$$\nu : (n : \mathsf{N})(M, O, P : \mathsf{Ob}(n))(p : \vdash_n M \sim_n O) \vdash_n \mathsf{app}(n, P, M) \sim_n \mathsf{app}(n, P, O)$$

$$\rho : (n : \mathsf{N})(M : \mathsf{Ob}(n)) \vdash_n M \sim_n M$$

$$\sigma : (n : \mathsf{N})(M, O : \mathsf{Ob}(n))(p : \vdash_n M \sim_n O) \vdash_n O \sim_n M$$

$$\tau : (n : \mathsf{N})(M, O, P : \mathsf{Ob}(n))(p : \vdash_n M \sim_n O)(q : \vdash_n O \sim_n P) \vdash_n M \sim_n P$$

$$
\begin{aligned}
\vdash_e :&(n : \mathsf{N})\\
&(C : (x : \mathsf{form}(n))(y : \vdash_n x)\mathsf{Set})\\
&(ek : (x : \mathsf{Ob}(n))(y : \mathsf{Ob}(n))C(n \sim_(,\mathsf{app}(n, \mathsf{app}(n, \mathsf{K}(n), x), y), x), \mathsf{Keq}(n, x, y)))\\
&(es :(x : \mathsf{Ob}(n))\\
&\quad (y : \mathsf{Ob}(n))\\
&\quad (z : \mathsf{Ob}(n))\\
&\quad C(\mathsf{app}(n, \mathsf{app}(n, \mathsf{app}(n, \mathsf{S}(n), x), y), z) \sim_n \mathsf{app}(n, \mathsf{app}(n, x, z), \mathsf{app}(n, y, z)),\\
&\qquad \mathsf{Seq}(n, x, y, z)))\\
&(emu :(x : \mathsf{Ob}(n))\\
&\qquad (y : \mathsf{Ob}(n))\\
&\qquad (z : \mathsf{Ob}(n))\\
&\qquad (t : \vdash_n x \sim_n y)\\
&\qquad (u : C(x \sim_n y), t))\\
&\qquad C(\mathsf{app}(n, x, z) \sim_n \mathsf{app}(n, y, z), \mu(n, x, y, z, t)))\\
&(enu :(x : \mathsf{Ob}(n))\\
&\qquad (y : \mathsf{Ob}(n))\\
&\qquad (z : \mathsf{Ob}(n))\\
&\qquad (t : \vdash_n x \sim_n y)\\
&\qquad (u : C(x \sim_n y, t))\\
&\qquad C(\mathsf{app}(n, z, x) \sim_n \mathsf{app}(n, z, y), \nu(n, x, y, z, t)))\\
&(ero : (x : \mathsf{Ob}(n))C(n \sim_(,x, x), \rho(n, x)))\\
&(esigma :(x : \mathsf{Ob}(n))\\
&\qquad\quad (y : \mathsf{Ob}(n))\\
&\qquad\quad (z : \vdash_n x \sim_n y)\\
&\qquad\quad (t : C(x \sim_n y, z))\\
&\qquad\quad C(y \sim_n x, \sigma(n, x, y, z)))\\
&(etau :(x : \mathsf{Ob}(n))\\
&\qquad (y : \mathsf{Ob}(n))\\
&\qquad (z : \mathsf{Ob}(n))\\
&\qquad (t : \vdash_n x \sim_n y)\\
&\qquad (u : \vdash_n y \sim_n z)\\
&\qquad (v : C(x \sim_n y, t))\\
&\qquad (w : C(y \sim_n z, u))\\
&\qquad C(x \sim_n z, \tau(n, x, y, z, t, u)))\\
&(f : \mathsf{form}(n))\\
&(p : \vdash_n f)\\
&C(f, p)
\end{aligned}
$$

$$
\begin{aligned}
&\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau,\\
&\quad app(n, \mathsf{app}(n, \mathsf{K}(n), x), y) \sim_n x, \mathsf{Keq}(n, x, y))\\
&\ = ek(x, y)
\end{aligned}
$$

$$
\begin{aligned}
&\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau,\\
&\quad \mathsf{app}(n, \mathsf{app}(n, \mathsf{app}(n, \mathsf{S}(n), x), y), z) \sim_n \mathsf{app}(n, \mathsf{app}(n, x, z), \mathsf{app}(n, y, z)), \mathsf{Seq}(n, x, y, z))\\
&\ = es(x, y, z)
\end{aligned}
$$

$$
\begin{aligned}
&\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, \mathsf{app}(n, x, z) \sim_n \mathsf{app}(n, y, z), \mu(n, x, y, z, t))\\
&\ = emu(x, y, z, t, \vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, x \sim_n y, t))
\end{aligned}
$$

$$\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, \mathsf{app}(n,z,x) \sim_n \mathsf{app}(n,z,y), \nu(n,x,y,z,t))$$
$$= enu(x,y,z,t,\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, x \sim_n y, t))$$

$$\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, x \sim_n x, \rho(n,x)) = ero(x)$$

$$\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, y \sim_n x, \sigma(n,x,y,z))$$
$$= esigma(x,y,z,\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, x \sim_n y, z))$$

$$\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, x \sim_n z, \tau(n,x,y,z,t,u))$$
$$= etau(x,y,z,t,u,$$
$$\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, x \sim_n y, t),$$
$$\vdash_e(n, C, ek, es, emu, enu, ero, esigma, etau, y \sim_n z, u))$$

The following two derived rules were proved:

$$\mathsf{Ieq} : (n : \mathsf{N})(x : \mathsf{Ob}(n))\vdash_n \mathsf{app}(n, \mathsf{I}(n), x) \sim_n x$$

$$\mathsf{Ieq} = [n,x]\tau(n,$$
$$\mathsf{app}(n, \mathsf{I}(n), x),$$
$$\mathsf{app}(n, \mathsf{app}(n, \mathsf{K}(n), x), \mathsf{app}(n, \mathsf{K}(n), x)),$$
$$x,$$
$$\mathsf{Seq}(n, \mathsf{K}(n), \mathsf{K}(n), x),$$
$$\mathsf{Keq}(n, x, \mathsf{app}(n, \mathsf{K}(n), x)))$$

$$\mu\nu : (n : \mathsf{N})(f,g,a,b : \mathsf{Ob}(n))(\vdash_n f \sim_n g)(\vdash_n a \sim_n b)\vdash_n \mathsf{app}(n,f,a) \sim_n \mathsf{app}(n,g,b)$$

$$\mu\nu = [n,f,g,a,b,h,h_1]\tau(n,$$
$$\mathsf{app}(n,f,a),$$
$$\mathsf{app}(n,g,a),$$
$$\mathsf{app}(n,g,b),$$
$$\mu(n,f,g,a,h),$$
$$\nu(n,a,b,g,h1))$$

The injection of a theory into an extension

$$\mathcal{O} : (m : \mathsf{N})(n : \mathsf{N})(p : m < n \ \vee m =_N n)(f : \mathsf{Ob}(m))\mathsf{Ob}(n)$$

$$\mathcal{O} = [m,n,p,f]$$
$$\mathsf{Ob}_e(m,$$
$$[x]\mathsf{Ob}(n),$$
$$\mathsf{K}(n),$$
$$\mathsf{S}(n),$$
$$[x]\mathsf{var}(n, \supset_e(\mathsf{IS}(m), \mathsf{IS}(n), \mathsf{ISInjection}\ (m,n,p), x)),$$
$$[x,y,hx,hy]\mathsf{app}(n,hx,hy),$$
$$f)$$

$$\mathcal{F} : (m : \mathsf{N})(n : \mathsf{N})(p : m < n \ \vee m =_N n)(f : \mathsf{form}(m))\mathsf{form}(n)$$

$$\mathcal{F} = [m,n,p,f]$$
$$\mathsf{form}_e(m, [f]\mathsf{form}(n), [x,y]\mathcal{O}(m,n,p,x) \sim_n \mathcal{O}(m,n,p,y), f)$$

$$\mathcal{T} : (m : \mathsf{N})(n : \mathsf{N})(p : m < n \ \vee m =_N n)(f : \mathsf{form}(m))(\vdash_m f)\vdash_n \mathcal{F}(m,n,p,f)$$

$$
\begin{aligned}
\mathcal{T} = [m,&n,p,f,h]\\
&\vdash_e(m,\\
&\quad [f_1,h]\vdash_n \mathcal{F}(m,n,p,f_1),\\
&\quad [x,y]\mathsf{Keq}(n,\mathcal{O}(m,n,p,x),\mathcal{O}(m,n,p,y)),\\
&\quad [x,y,z]\mathsf{Seq}(n,\mathcal{O}(m,n,p,x),\mathcal{O}(m,n,p,y),\mathcal{O}(m,n,p,z)),\\
&\quad [x,y,z,t,h_1]\mu(n,\mathcal{O}(m,n,p,x),\mathcal{O}(m,n,p,y),\mathcal{O}(m,n,p,z),h_1),\\
&\quad [x,y,z,t,h_1]\nu(n,\mathcal{O}(m,n,p,x),\mathcal{O}(m,n,p,y),\mathcal{O}(m,n,p,z),h_1),\\
&\quad [x]\rho(n,\mathcal{O}(m,n,p,x)),\\
&\quad [x,y,z,h_1]\sigma(n,\mathcal{O}(m,n,p,x),\mathcal{O}(m,n,p,y),h_1),\\
&\quad [x,y,z,t,u,h_1,h_2]\tau(n,\mathcal{O}(m,n,p,x),\mathcal{O}(m,n,p,y),\mathcal{O}(m,n,p,z),h_1,h_2),\\
&\quad f,\\
&\quad h)
\end{aligned}
$$

The definition of abstraction of one variable

The definition consists of an application of the rule Ob_e. We first show the terms that correspond to the premises of that rule and then plug them into it to build the definition.

$$\mathsf{KAbstAlg} : (n : \mathsf{N})\mathsf{Ob}(n)$$

$$\mathsf{KAbstAlg} = [n]\mathsf{app}(n,\mathsf{K}(n),\mathsf{K}(n))$$

$$\mathsf{SAbstAlg} : (n : \mathsf{N})\mathsf{Ob}(n)$$

$$\mathsf{SAbstAlg} = [n]\mathsf{app}(n,\mathsf{K}(n),\mathsf{S}(n))$$

$$\mathsf{varAbstAlg} : (n : \mathsf{N})(x : \mathsf{IS}(\mathsf{s}(n)))\mathsf{Ob}(n)$$

$$\mathsf{varAbstAlg} = [n,x]\mathsf{IS}_e(n,[x]\mathsf{Ob}(n),\mathsf{I}(n),[i]\mathsf{app}(n,\mathsf{K}(n),\mathsf{var}(n,i)),x)$$

$$\mathsf{appAbstAlg} : (n : \mathsf{N})(x : \mathsf{Ob}(\mathsf{s}(n)))(y : \mathsf{Ob}(\mathsf{s}(n)))(\mathsf{Ob}(n))(\mathsf{Ob}(n))\mathsf{Ob}(n)$$

$$\mathsf{appAbstAlg} = [n,x,y,h_1,h_2]\mathsf{app}(n,\mathsf{app}(n,\mathsf{S}(n),h_1),h_2)$$

$$\mathsf{AbstAlg} : (n : \mathsf{N})(\mathsf{Ob}(\mathsf{s}(n)))\mathsf{Ob}(n)$$

$$
\begin{aligned}
\mathsf{AbstAlg} = [n,h]\mathsf{Ob}_e(&\mathsf{s}(n),\\
&[h_1]\mathsf{Ob}(n),\\
&\mathsf{KAbstAlg}(n),\\
&\mathsf{SAbstAlg}(n),\\
&[x]\mathsf{varAbstAlg}(n,x),\\
&[x,y,h_1,h_2]\mathsf{appAbstAlg}(n,x,y,h_1,h_2),\\
&h)
\end{aligned}
$$

The proof of the combinatorial completeness

We first abbreviate the statement with the set $\mathsf{CombComp}$ and then prove it using Ob_e. As in the definition of $\mathsf{AbstAlg}$ we prove all the cases corresponding to the premises and then plug them into the rule. We will not write all the arguments to the injections of a theory into another one, we will only keep what is injected and the extensions involved. Thus, $\mathcal{O}(m,n,p,f)$ will be abbreviated with $\mathcal{O}(m,n,f)$.

$\mathsf{CombComp} : (n : \mathsf{N})(X : \mathsf{Ob}(\mathsf{s}(n)))\mathsf{Set}$

$\mathsf{CombComp} = [n, X]\vdash_{\mathsf{s}(n)} \mathsf{app}(\mathsf{s}(n), \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, X)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))) \sim_{\mathsf{s}(n)} X$

$\mathsf{KCombComp} : (n : \mathsf{N})\mathsf{CombComp}(n, \mathsf{K}(\mathsf{s}(n)))$

$\mathsf{KCombComp} = [n]\mathsf{Keq}(\mathsf{s}(n), \mathsf{K}(\mathsf{s}(n)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n)))$

$\mathsf{SCombComp} : (n : \mathsf{N})\mathsf{CombComp}(n, \mathsf{S}(\mathsf{s}(n)))$

$\mathsf{SCombComp} = [n]\mathsf{Keq}(\mathsf{s}(n), \mathsf{S}(\mathsf{s}(n)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n)))$

$\mathsf{varCombComp} : (n : \mathsf{N})(x : \mathsf{IS}(\mathsf{s}(n)))\mathsf{CombComp}(n, \mathsf{var}(\mathsf{s}(n), x))$

$$
\begin{aligned}
\mathsf{varCombComp} = [n, x]\mathsf{IS}_e(n,\ & \\
& [h]\mathsf{CombComp}(n, \mathsf{var}(\mathsf{s}(n), h)), \\
& \mathsf{Ieq}(\mathsf{s}(n), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& [i]\mathsf{Keq}(\mathsf{s}(n), \mathsf{var}(\mathsf{s}(n), \mathsf{inject}(n, i)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& x)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{appCombComp} :\ & (n : \mathsf{N}) \\
& (f : \mathsf{Ob}(\mathsf{s}(n))) \\
& (g : \mathsf{Ob}(\mathsf{s}(n))) \\
& (fh : \mathsf{CombComp}(n, f)) \\
& (gh : \mathsf{CombComp}(n, g)) \\
& \mathsf{CombComp}(n, \mathsf{app}(\mathsf{s}(n), f, g))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{appCombComp} = [n, f, g, fh, gh]\tau(\mathsf{s}(n),\ & \\
& \mathsf{app}(\mathsf{s}(n), \\
& \quad \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, \mathsf{app}(\mathsf{s}(n), f, g))), \\
& \quad \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& \mathsf{app}(\mathsf{s}(n), \\
& \quad \mathsf{app}(\mathsf{s}(n), \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, f)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& \quad \mathsf{app}(\mathsf{s}(n), \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, g)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n)))), \\
& \mathsf{app}(\mathsf{s}(n), f, g), \\
& \mathsf{Seq}(\mathsf{s}(n), \\
& \quad \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, f)), \\
& \quad \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, g)), \\
& \quad \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& \mu\nu(\mathsf{s}(n), \\
& \quad \mathsf{app}(Succ(n), \\
& \quad\quad \mathcal{O}(n, Succ(n), \mathsf{AbstAlg}(n, f)), \\
& \quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& \quad f, \\
& \quad \mathsf{app}(\mathsf{s}(n), \mathcal{O}(n, \mathsf{s}(n), \mathsf{AbstAlg}(n, g)), \mathsf{var}(\mathsf{s}(n), \mathsf{put}(n))), \\
& \quad g, \\
& \quad fh, \\
& \quad gh))
\end{aligned}
$$

$\mathsf{ObCC} : (n : \mathsf{N})(X : \mathsf{Ob}(\mathsf{s}(n)))\mathsf{CombComp}(n, X)$

$$
\begin{aligned}
\mathsf{ObCC} = [n, X]\mathsf{Ob}_e(&\mathsf{s}(n),\\
&[X]\mathsf{CombComp}(n, X),\\
&\mathsf{KCombComp}(n),\\
&\mathsf{SCombComp}(n),\\
&[x]\mathsf{varCombComp}(n, x),\\
&[x, y, h_1, h_2]\mathsf{appCombComp}(n, x, y, h_1, h_2),\\
&X)
\end{aligned}
$$

The abstraction of all the variables of an extension

$$\mathsf{EAbstAlg} : (n : \mathsf{N})\mathsf{Ob}(n) \supset \mathsf{Ob}(0)$$

$$
\begin{aligned}
\mathsf{EAbstAlg} = [n]\mathsf{natrec}(&[n]\mathsf{Ob}(n) \supset \mathsf{Ob}(0),\\
&\lambda(\mathsf{Ob}(0), \mathsf{Ob}(0), [h]h),\\
&[x, h]\lambda(\mathsf{Ob}(\mathsf{s}(x)), \mathsf{Ob}(0), [h_1]\supset_e(\mathsf{Ob}(x), \mathsf{Ob}(0), h, \mathsf{AbstAlg}(x, h_1))),\\
&n)
\end{aligned}
$$

We need the definition of the application to all the indeterminates of an extension

$$\mathsf{Eapp} : (n : \mathsf{N})\mathsf{Ob}(0) \supset \mathsf{Ob}(n)$$

$$
\begin{aligned}
\mathsf{Eapp} = [n]\mathsf{natrec}(&[n]\mathsf{Ob}(0) \supset \mathsf{Ob}(n),\\
&\lambda(\mathsf{Ob}(0), \mathsf{Ob}(0), [h]h),\\
&[x, h]\lambda(\mathsf{Ob}(0), \mathsf{Ob}(\mathsf{s}(x)), [h_1]\mathsf{app}(\mathsf{s}(x),\\
&\qquad\qquad \mathcal{O}(x, \mathsf{s}(x), \supset_e(\mathsf{Ob}(0), \mathsf{Ob}(x), h, h_1)),\\
&\qquad\qquad \mathsf{var}(\mathsf{s}(x), \mathsf{put}(x)))),\\
&n)
\end{aligned}
$$

the two following terms make the proof more readable:

$$\mathsf{app}^* : (n : \mathsf{N})(\mathsf{Ob}(0))\mathsf{Ob}(n)$$

$$\mathsf{app}^* = [n, h]\supset_e(\mathsf{Ob}(0), \mathsf{Ob}(n), \mathsf{Eapp}(n), h)$$

$$\mathsf{AbstAlg}^* : (n : \mathsf{N})(\mathsf{Ob}(n))\mathsf{Ob}(0)$$

$$\mathsf{AbstAlg}^* = [n, h]\supset_e(\mathsf{Ob}(n), \mathsf{Ob}(0), \mathsf{EAbstAlg}(n), h)$$

$$\mathsf{ObCC}^* : (n : \mathsf{N})\Pi(\mathsf{Ob}(n), [X]\vdash_n \mathsf{app}^*(n, \mathsf{AbstAlg}^*(n, X)) \sim_n X)$$

$$
\begin{array}{l}
\mathsf{ObCC}^* = [n]\mathsf{natrec}([n]\Pi(\mathsf{Ob}(n), [X]\vdash_n \mathsf{app}^*(n, \mathsf{AbstAlg}^*(n, X)) \sim_n X), \\
\qquad \Lambda(\mathsf{Ob}(0), [h_0]\vdash_0 \mathsf{app}^*(0, \mathsf{AbstAlg}^*(0, h_0)) \sim_0 h_0, [x_1]\rho(0, x_1)), \\
\qquad [x, h]\Lambda(\mathsf{Ob}(\mathsf{s}(x)), \\
\qquad\qquad [h_0]\vdash_{\mathsf{s}(x)} \mathsf{app}^*(\mathsf{s}(x), \mathsf{AbstAlg}^*(\mathsf{s}(x), h_0)) \sim_{\mathsf{s}(x)} h_0, \\
\qquad\qquad [x_1]\tau(\mathsf{s}(x), \\
\qquad\qquad\qquad \mathsf{app}(\mathsf{s}(x), \\
\qquad\qquad\qquad\quad \mathcal{O}(x, \mathsf{s}(x), \mathsf{app}^*(x, \mathsf{AbstAlg}^*(\mathsf{s}(x), x_1))), \\
\qquad\qquad\qquad\quad \mathsf{var}(\mathsf{s}(x), \mathsf{put}(x))), \\
\qquad\qquad\qquad \mathsf{app}(\mathsf{s}(x), \\
\qquad\qquad\qquad\quad \mathcal{O}(x, \mathsf{s}(x), \mathsf{AbstAlg}(x, x_1)), \\
\qquad\qquad\qquad\quad \mathsf{var}(\mathsf{s}(x), \mathsf{put}(x))), \\
\qquad\qquad\qquad x_1, \\
\qquad\qquad\qquad \mu(\mathsf{s}(x), \\
\qquad\qquad\qquad\; \mathcal{O}(x, \mathsf{s}(x), \mathsf{app}^*(x, \mathsf{AbstAlg}^*(\mathsf{s}(x), x_1))), \\
\qquad\qquad\qquad\; \mathcal{O}(x, \mathsf{s}(x), \mathsf{AbstAlg}(x, x_1)), \\
\qquad\qquad\qquad\; \mathsf{var}(\mathsf{s}(x), \mathsf{put}(x)), \\
\qquad\qquad\qquad\; \mathcal{T}(x, \\
\qquad\qquad\qquad\quad\; \mathsf{s}(x), \\
\qquad\qquad\qquad\quad\; \mathsf{app}^*(x, \mathsf{AbstAlg}^*(x, \mathsf{AbstAlg}(x, x_1))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \sim_x \mathsf{AbstAlg}(x, x_1), \\
\qquad\qquad\qquad\quad\; \Pi_e(\mathsf{Ob}(x), \\
\qquad\qquad\qquad\qquad\; [X]\vdash_x \mathsf{app}^*(x, \mathsf{AbstAlg}^*(x, X)) \sim_x X, \\
\qquad\qquad\qquad\qquad\; h, \\
\qquad\qquad\qquad\qquad\; \mathsf{AbstAlg}(x, x_1)))), \\
\qquad\qquad\qquad \mathsf{ObCC}(x, x_1))), \\
\qquad n)
\end{array}
$$

Typed Combinators

Types

types : Set

atoms : Set

at : (a : atoms)types

$\rightarrow$: (α : types)(β : types)types

$$
\begin{array}{lll}
\mathsf{types}_e & : & (P : (\mathsf{types})\mathsf{Set}) \\
 & & (atc : (a : \mathsf{atoms})P(\mathsf{at}(a))) \\
 & & (arrc : (\alpha : \mathsf{types})(\beta : \mathsf{types})(\alpha_h : P(\alpha))(\beta_h : P(\beta))P(\alpha \rightarrow \beta)) \\
 & & (\alpha : \mathsf{types}) \\
 & & P(\alpha)
\end{array}
$$

Type assignments

TA : (N)Set

nil : TA(0)

$\mathsf{cons} : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})\mathsf{TA}(\mathsf{s}(n))$

$$\begin{array}{lll}\mathsf{TA}_e & : & (P : (n : \mathsf{N})(\mathsf{TA}(n))\mathsf{Set}) \\ & & (nilc : P(0, \mathsf{nil})) \\ & & (consc : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(P(n, ta))P(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha))) \\ & & (n : \mathsf{N}) \\ & & (ta : \mathsf{TA}(n)) \\ & & P(n, ta)\end{array}$$

The typing of a variable

The sets corresponding to the typed combinators. First the set coding the typing of a variable in a type assignement:

$\mathsf{VarType} : (n : \mathsf{N})(ta : \mathsf{TA}(n))(x : \mathsf{IS}(n))(\alpha : \mathsf{types})\mathsf{Set}$

$\mathsf{PutType} : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})\mathsf{VarType}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \mathsf{put}(n), \alpha)$

$$\begin{array}{lll}\mathsf{InjectType} & : & (n : \mathsf{N}) \\ & & (ta : \mathsf{TA}(n)) \\ & & (\alpha : \mathsf{types}) \\ & & (\beta : \mathsf{types}) \\ & & (y : \mathsf{IS}(n)) \\ & & (p : \mathsf{VarType}(n, ta, y, \alpha)) \\ & & \mathsf{VarType}(\mathsf{s}(n), \mathsf{cons}(n, ta, \beta), \mathsf{inject}(n, y), \alpha)\end{array}$$

$$\begin{array}{lll}\mathsf{VarType}_e & : & (n : \mathsf{N}) \\ & & (ta : \mathsf{TA}(n)) \\ & & \begin{array}{lll}(P & : & (x : \mathsf{IS}(\mathsf{s}(n))) \\ & & (\alpha : \mathsf{types}) \\ & & (\beta : \mathsf{types}) \\ & & (p : \mathsf{VarType}(\mathsf{s}(n), \mathsf{cons}(n, ta, \beta), x, \alpha)) \\ & & \mathsf{Set})\end{array} \\ & & (e : (\alpha : \mathsf{types})P(\mathsf{put}(n), \alpha, \alpha, \mathsf{PutType}(n, ta, \alpha))) \\ & & \begin{array}{lll}(d & : & (\alpha : \mathsf{types}) \\ & & (\beta : \mathsf{types}) \\ & & (y : \mathsf{IS}(n)) \\ & & (p : \mathsf{VarType}(n, ta, y, \alpha)) \\ & & P(\mathsf{inject}(n, y), \alpha, \beta, \mathsf{InjectType}(n, ta, \alpha, \beta, y, p)))\end{array} \\ & & (x : \mathsf{IS}(\mathsf{s}(n))) \\ & & (\alpha : \mathsf{types}) \\ & & (\beta : \mathsf{types}) \\ & & (p : \mathsf{VarType}(\mathsf{s}(n), \mathsf{cons}(n, ta, \beta), x, \alpha)) \\ & & P(x, \alpha, \beta, p)\end{array}$$

$\mathsf{VarType}_e(n, ta, P, e, d, \mathsf{put}(n), \alpha, \alpha, \mathsf{PutType}(n, ta, \alpha)) = e(\alpha)$

$\mathsf{VarType}_e(n, ta, P, e, d, \mathsf{inject}(n, y), \alpha, \beta, \mathsf{InjectType}(n, ta, \alpha, \beta, y, p)) = d(\alpha, \beta, y, p)$

The objects

$$\mathsf{ob} : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})\mathsf{Set}$$

$$\begin{array}{lll} \mathsf{K} & : & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(\beta : \mathsf{types}) \\ & & \mathsf{ob}(n, ta, \alpha \to \beta \to \alpha) \end{array}$$

$$\begin{array}{lll} \mathsf{S} & : & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(\beta : \mathsf{types})(\gamma : \mathsf{types}) \\ & & \mathsf{ob}(n, ta, (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma) \end{array}$$

$$\begin{array}{lll} \mathsf{var} & : & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(x : \mathsf{IS}(n))(p : \mathsf{VarType}(n, ta, x, \alpha)) \\ & & \mathsf{ob}(n, ta, \alpha) \end{array}$$

$$\begin{array}{lll} \mathsf{app} & : & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(\beta : \mathsf{types})(f : \mathsf{ob}(n, ta, \alpha \to \beta))(a : \mathsf{ob}(n, ta, \alpha)) \\ & & \mathsf{ob}(n, ta, \beta) \end{array}$$

$$\begin{array}{lll} \mathsf{ob}_e & : & (n : \mathsf{N}) \\ & & (ta : \mathsf{TA}(n)) \\ & & (P : (\alpha : \mathsf{types})(\mathsf{ob}(n, ta, \alpha))\mathsf{Set}) \\ & & (kc : (\alpha : \mathsf{types})(\beta : \mathsf{types})P(\alpha \to \beta \to \alpha), \mathsf{K}(n, ta, \alpha, \beta))) \\ & & (\begin{array}{lll} sc & : & (\alpha : \mathsf{types})(\beta : \mathsf{types})(\gamma : \mathsf{types}) \\ & & P((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathsf{S}(n, ta, \alpha, \beta, \gamma))) \end{array} \\ & & (varc : (\alpha : \mathsf{types})(x : \mathsf{IS}(n))(p : \mathsf{VarType}(n, ta, x, \alpha))P(\alpha, Var(n, ta, \alpha, x, p))) \\ & & (\begin{array}{lll} appc & : & (\alpha : \mathsf{types})(\beta : \mathsf{types}) \\ & & (f : \mathsf{ob}(n, ta, \alpha \to \beta)) \\ & & (a : \mathsf{ob}(n, ta, \alpha)) \\ & & (fh : P(\alpha \to \beta, f)) \\ & & (ah : P(\alpha, a))P(\beta, \mathsf{app}(n, ta, \alpha, \beta, f, a))) \end{array} \\ & & (\gamma : \mathsf{types}) \\ & & (t : \mathsf{ob}(n, ta, \gamma)) \\ & & P(\gamma, t) \end{array}$$

$$\mathsf{ob}_e(n, ta, P, kc, sc, varc, appc, \alpha \to \beta \to \alpha, \mathsf{K}(n, ta, \alpha, \beta)) = kc(\alpha, \beta)$$

$$\begin{array}{l} \mathsf{ob}_e(n, ta, P, kc, sc, varc, appc, (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathsf{S}(n, ta, \alpha, \beta, \gamma)) \\ = sc(\alpha, \beta, \gamma) \end{array}$$

$$\mathsf{ob}_e(n, ta, P, kc, sc, varc, appc, \alpha, \mathsf{var}(n, ta, \alpha, x, p)) = varc(\alpha, x, p)$$

$$\begin{array}{l} \mathsf{ob}_e(n, ta, P, kc, sc, varc, appc, \beta, \mathsf{app}(n, ta, \alpha, \beta, f, a)) = \\ appc(\alpha, \beta, f, a, \\ \qquad \mathsf{ob}_e(n, ta, P, kc, sc, varc, appc, \alpha \to \beta, f), \\ \qquad \mathsf{ob}_e(n, ta, P, kc, sc, varc, appc, \alpha, a)) \end{array}$$

We define the identity combinator:

$$\mathsf{I} : (\alpha : \mathsf{types})\mathsf{ob}(n, ta, \alpha \to \alpha)$$

$$
\begin{aligned}
\mathsf{I} = [\alpha]\mathsf{app}(\quad & n, \\
& ta, \\
& \alpha \to \alpha \to \alpha, \\
& \alpha \to \alpha, \\
& \mathsf{app}(\quad n, \\
& \qquad ta, \\
& \qquad \alpha \to \alpha \to \alpha \to \alpha, \\
& \qquad \alpha \to \alpha \to \alpha \to \alpha \to \alpha, \\
& \qquad \mathsf{S}(n, ta, \alpha, \alpha \to \alpha, \alpha), \\
& \qquad \mathsf{K}(n, ta, \alpha, \alpha \to \alpha)), \\
& \mathsf{K}(n, ta, \alpha, \alpha))
\end{aligned}
$$

The injection of objects

The Injection of the objects of one theory into an extension with one more variable is realized by the term:

$$\mathsf{ObInj} : (n : \mathsf{N}; ta : \mathsf{TA}(n); \gamma : \mathsf{types}; \delta : \mathsf{types}; t : \mathsf{ob}(n, ta, \gamma))\mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \delta), \gamma)$$

which is defined as:

$$
\begin{aligned}
\mathsf{ObInj} = [n, ta, \gamma, \delta, t]\mathsf{ob}_e(\quad & n, \\
& ta, \\
& [\gamma_1, t_1]\mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, delta), \gamma_1), \\
& [\alpha_1, \beta_1]\mathsf{K}(\mathsf{s}(n), \mathsf{cons}(n, ta, \delta), \alpha_1, \beta_1), \\
& [\alpha_1, \beta_1, \gamma_1]\mathsf{S}(\quad \mathsf{s}(n), \\
& \qquad \mathsf{cons}(n, ta, \delta), \\
& \qquad \alpha_1, \\
& \qquad \beta_1, \\
& \qquad \gamma_1), \\
& [\alpha_1, x_1, p_1]\mathsf{var}(\quad \mathsf{s}(n), \\
& \qquad \mathsf{cons}(n, ta, \delta), \\
& \qquad \alpha_1, \\
& \qquad \mathsf{inject}(n, x_1), \\
& \qquad \mathsf{InjectType}(n, ta, \alpha_1, \delta, x_1, p_1)), \\
& [\alpha_1, \beta_1, f_1, a_1, h, h_1]\mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \delta), \alpha_1, \beta_1, h, h_1), \\
& \gamma, \\
& t)
\end{aligned}
$$

The formulae

$$\mathsf{form} : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})\mathsf{Set}$$

$$\sim : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(t : \mathsf{ob}(n, ta, \alpha))(u : \mathsf{ob}(n, ta, \alpha))\mathsf{form}(n, ta, \alpha)$$

The theorems

$$\vdash : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(eq : \mathsf{form}(n, ta, \alpha))\mathsf{Set}$$

In what follows we will use $t \sim_{n,ta} u$ for $\sim (n, ta, \alpha, t, u)$ and $\vdash_{n,ta} eq$ for $\vdash (n, ta, \alpha, eq)$ as abbreviations.

$$
\begin{aligned}
\mathsf{Keq} \quad : \quad & (n : \mathsf{N}) \\
& (ta : \mathsf{TA}(n)) \\
& (\alpha : \mathsf{types}) \\
& (\beta : \mathsf{types}) \\
& (a : \mathsf{ob}(n, ta, \alpha)) \\
& (b : \mathsf{ob}(n, ta, \beta)) \\
& \vdash_{n,ta} \mathsf{app}(n, ta, \beta, \alpha, \mathsf{app}(n, ta, \alpha, \beta \to \alpha, \mathsf{K}(n, ta, \alpha, \beta), a), b) \sim_{n,ta} a
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{Seq} \quad : \quad & (n : \mathsf{N}) \\
& (ta : \mathsf{TA}(n)) \\
& (\alpha : \mathsf{types}) \\
& (\beta : \mathsf{types}) \\
& (\gamma : \mathsf{types}) \\
& (f : \mathsf{ob}(n, ta, \alpha \to \beta \to \gamma)) \\
& (g : \mathsf{ob}(n, ta, \alpha \to \beta)) \\
& (a : \mathsf{ob}(n, ta, \alpha)) \\
& \vdash_{n,ta} \mathsf{app}(\mathsf{app}(\mathsf{app}(\mathsf{S}(n, ta, \alpha, \beta, \gamma), f), g), a) \sim_{n,ta} \mathsf{app}(\mathsf{app}(f, a), \mathsf{app}(g, a))
\end{aligned}
$$

where I have removed all the type and context information from the applications to make it more readable, but it can be restored by looking at the rules of object formation.

$$
\begin{aligned}
\mu \quad : \quad & (n : \mathsf{N}) \\
& (ta : \mathsf{TA}(n)) \\
& (\alpha : \mathsf{types}) \\
& (\beta : \mathsf{types}) \\
& (f : \mathsf{ob}(n, ta, \alpha \to \beta)) \\
& (g : \mathsf{ob}(n, ta, \alpha \to \beta)) \\
& (a : \mathsf{ob}(n, ta, \alpha)) \\
& (\vdash_{n,ta} f \sim_{n,ta} g) \\
& \vdash_{n,ta} \mathsf{app}(n, ta, \alpha, \beta, f, a) \sim_{n,ta} \mathsf{app}(n, ta, \alpha, \beta, g, a)
\end{aligned}
$$

$$
\begin{aligned}
\nu \quad : \quad & (n : \mathsf{N}) \\
& (ta : \mathsf{TA}(n)) \\
& (\alpha : \mathsf{types}) \\
& (\beta : \mathsf{types}) \\
& (a : \mathsf{ob}(n, ta, \alpha)) \\
& (b : \mathsf{ob}(n, ta, \alpha)) \\
& (f : \mathsf{ob}(n, ta, \alpha \to \beta)) \\
& (\vdash_{n,ta} a \sim_{n,ta} b) \\
& \vdash_{n,ta} \mathsf{app}(n, ta, \alpha, \beta, f, a) \sim_{n,ta} \mathsf{app}(n, ta, \alpha, \beta, f, b)
\end{aligned}
$$

$$\rho : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(t : \mathsf{ob}(n, ta, \alpha)) \vdash_{n,ta} t \sim_{n,ta} t$$

$$
\begin{aligned}
\sigma : \quad & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(t : \mathsf{ob}(n, ta, \alpha))(u : \mathsf{ob}(n, ta, \alpha)) \\
& (\vdash_{n,ta} t \sim_{n,ta} u) \vdash_{n,ta} u \sim_{n,ta} t
\end{aligned}
$$

$$
\begin{aligned}
\tau : \quad & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(t : \mathsf{ob}(n, ta, \alpha))(u : \mathsf{ob}(n, ta, \alpha))(v : \mathsf{ob}(n, ta, \alpha)) \\
& (\vdash_{n,ta} t \sim_{n,ta} u)(\vdash_{n,ta} u \sim_{n,ta} v) \vdash_{n,ta} t \sim_{n,ta} v
\end{aligned}
$$

The following derived rules were proved, which are then used in the proof of combinatorial completeness:

$$
\begin{aligned}
\mu\nu : (\quad & n : \mathsf{N}; ta : \mathsf{TA}(n);\\
& \alpha : \mathsf{types}; \beta : \mathsf{types};\\
& f : \mathsf{ob}(n, ta, \alpha \to \beta);\\
& g : \mathsf{ob}(n, ta, \alpha \to \beta);\\
& a : \mathsf{ob}(n, ta, \alpha);\\
& b : \mathsf{ob}(n, ta, \alpha);\\
& \vdash_{n,ta} f \sim_{n,ta} g;\\
& \vdash_{n,ta} a \sim_{n,ta} b)\\
& \vdash_{n,ta} \mathsf{app}(n, ta, \alpha, \beta, f, a) \sim_{n,ta} \mathsf{app}(n, ta, \alpha, \beta, g, b)
\end{aligned}
$$

$$
\begin{aligned}
\mu\nu = [n, ta, \alpha, \beta, f, g, a, b, h, h1]\tau(& n,\\
& ta,\\
& \beta,\\
& \mathsf{app}(n, ta, \alpha, \beta, f, a),\\
& \mathsf{app}(n, ta, \alpha, \beta, g, a),\\
& \mathsf{app}(n, ta, \alpha, \beta, g, b),\\
& \mu(n, ta, \alpha, \beta, f, g, a, h),\\
& \nu(n, ta, \alpha, \beta, a, b, g, h1))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{IdEq} :\quad & (n : \mathsf{N}; ta : \mathsf{TA}(n); \alpha : \mathsf{types}; a : \mathsf{ob}(n, ta, \alpha))\\
& \vdash_{n,ta} \mathsf{app}(n, ta, \alpha, \alpha, \mathsf{I}(n, ta, \alpha), a) \sim_{n,ta} a
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{IdEq} = [n, ta, \alpha, a]\tau(& n,\\
& ta,\\
& \alpha,\\
& \mathsf{app}(n, ta, \alpha, \alpha, \mathsf{I}(n, ta, \alpha), a),\\
& \mathsf{app}(\quad n, ta, \alpha \to \alpha, \alpha,\\
& \qquad \mathsf{app}(n, ta, \alpha, \alpha \to \alpha \to \alpha, \mathsf{K}(n, ta, \alpha, \alpha \to \alpha), a),\\
& \qquad \mathsf{app}(n, ta, \alpha, (\to \alpha, \alpha), \mathsf{K}(n, ta, \alpha, \alpha), a)),\\
& a,\\
& \mathsf{Seq}(n, ta, \alpha, \alpha \to \alpha, \alpha, \mathsf{K}(n, ta, \alpha, \alpha \to \alpha), \mathsf{K}(n, ta, \alpha, \alpha), a),\\
& \mathsf{Keq}(n, ta, \alpha, \alpha \to \alpha, a, \mathsf{app}(n, ta, \alpha, \alpha \to \alpha, \mathsf{K}(n, ta, \alpha, \alpha), a)))
\end{aligned}
$$

Definition of Abstraction

The definition is by using the elimination rule for objects of a given type in a given extension. We define separately each of the premisses of this rule and then plug these definitions into the main one. The typings of these terms are:

$$
\begin{aligned}
\mathsf{KAbstAlg} :\quad & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(\gamma : \mathsf{types})(\delta : \mathsf{types})\\
& \mathsf{ob}(n, ta, \alpha \to \gamma \to \delta \to \gamma)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{SAbstAlg} :\quad & (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(\gamma : \mathsf{types})(\delta : \mathsf{types})(\kappa : \mathsf{types})\\
& \mathsf{ob}(n, ta, \alpha \to (\gamma \to \delta \to \kappa) \to (\gamma \to \delta) \to \gamma \to \kappa)
\end{aligned}
$$

$$\begin{array}{ll}\mathsf{VarAbstAlg}: & (n:\mathsf{N})(ta:\mathsf{TA}(n))(\alpha:\mathsf{types})(\beta:\mathsf{types})\\ & (x:\mathsf{IS}(\mathsf{s}(n)))\\ & (p:\mathsf{VarType}(\mathsf{s}(n),\mathsf{cons}(n,ta,\alpha),x,\beta))\\ & \mathsf{ob}(n,ta,\alpha\rightarrow\beta)\end{array}$$

$$\begin{array}{ll}\mathsf{AppAbstAlg}: & (n:\mathsf{N})(ta:\mathsf{TA}(n))(\alpha:\mathsf{types})(\gamma:\mathsf{types})(\delta:\mathsf{types})\\ & (f:\mathsf{ob}(\mathsf{s}(n),\mathsf{cons}(n,ta,\alpha),\gamma\rightarrow\delta))\\ & (a:\mathsf{ob}(\mathsf{s}(n),\mathsf{cons}(n,ta,\alpha),\gamma))\\ & (fh:\mathsf{ob}(n,ta,\alpha\rightarrow\gamma\rightarrow\delta))\\ & (ah:\mathsf{ob}(n,ta,\alpha\rightarrow\gamma))\\ & \mathsf{ob}(n,ta,\alpha\rightarrow\delta)\end{array}$$

And the typing of the abstraction algorithm:

$$\begin{array}{ll}\mathsf{AbstAlg}: & (n:\mathsf{N})(ta:\mathsf{TA}(n))(\alpha:\mathsf{types})(\beta:\mathsf{types})\\ & (X:\mathsf{ob}(\mathsf{s}(n),\mathsf{cons}(n,ta,\alpha),\beta))\\ & \mathsf{ob}(n,ta,\alpha\rightarrow\beta)\end{array}$$

Now the terms that define these constants:

$$\begin{array}{l}\mathsf{KAbstAlg}=[n,ta,\alpha,\gamma,\delta]\mathsf{app}(n,\\ \quad ta,\\ \quad \gamma\rightarrow\delta\rightarrow\gamma,\\ \quad \alpha\rightarrow\gamma\rightarrow\delta\rightarrow\gamma,\\ \quad \mathsf{K}(n,ta,\gamma\rightarrow\delta\rightarrow\gamma,\alpha),\\ \quad \mathsf{K}(n,ta,\gamma,\delta))\end{array}$$

$$\begin{array}{l}\mathsf{SAbstAlg}=[n,ta,\alpha,\gamma,\delta,\kappa]\mathsf{app}(n,\\ \quad ta,\\ \quad (\gamma\rightarrow\delta\rightarrow\kappa)\rightarrow(\gamma\rightarrow\delta)\rightarrow\gamma\rightarrow\kappa,\\ \quad \alpha\rightarrow(\gamma\rightarrow\delta\rightarrow\kappa)\rightarrow(\gamma\rightarrow\delta)\rightarrow\gamma\rightarrow\kappa,\\ \quad \mathsf{K}(n,ta,(\gamma\rightarrow\delta\rightarrow\kappa)\rightarrow(\gamma\rightarrow\delta)\rightarrow\gamma\rightarrow\kappa,\alpha),\\ \quad \mathsf{S}(n,ta,\gamma,\delta,\kappa))\end{array}$$

$$\begin{array}{l}\mathsf{VarAbstAlg}=[n,ta,\alpha,\beta,x,p]\mathsf{VarType}_e(n,\\ \quad ta,\\ \quad [x_1,\alpha_1,\alpha_2,p_1]\mathsf{ob}(n,ta,\alpha_2\rightarrow\alpha_1),\\ \quad [\alpha_1]\mathsf{I}(\alpha_1)\\ \quad [\alpha_1,\beta_1,y_1,p_1]\mathsf{app}(\quad n,\\ \qquad ta,\\ \qquad \alpha_1,\\ \qquad \beta_1\rightarrow\alpha_1,\\ \qquad \mathsf{K}(n,ta,\alpha_1,\beta_1),\\ \qquad \mathsf{var}(n,ta,\alpha_1,y_1,p_1)),\\ \quad x,\\ \quad \beta,\\ \quad \alpha,\\ \quad p)\end{array}$$

$$\begin{aligned}\mathsf{AppAbstAlg} = [n, ta, \alpha, \gamma, \delta, f, a, fh, ah]\mathsf{app}(&n,\\ &ta,\\ &\alpha \to \gamma,\\ &\alpha \to \delta,\\ &\mathsf{app}(\quad n,\\ &\qquad ta,\\ &\qquad \alpha \to \gamma \to \delta,\\ &\qquad \alpha \to \gamma \to \alpha \to \delta,\\ &\qquad \mathsf{S}(n, ta, \alpha, \gamma, \delta),\\ &\qquad fh),\\ &ah)\end{aligned}$$

And the term for the abstraction algorithm, which uses the terms above:

$$\begin{aligned}\mathsf{AbstAlg} = [n, ta, \alpha, \beta, X]\mathsf{ob}_e(&\mathsf{s}(n),\\ &\mathsf{cons}(n, ta, \alpha),\\ &[\beta_1, X_1]\mathsf{ob}(n, ta, \alpha \to \beta_1),\\ &[\alpha_1, \beta_1]\mathsf{KAbstAlg}(n, ta, \alpha, \alpha_1, \beta_1),\\ &[\alpha_1, \beta_1, \gamma_1]\mathsf{SAbstAlg}(n, ta, \alpha, \alpha_1, \beta_1, \gamma_1),\\ &[\alpha_1, x_1, p_1]\mathsf{VarAbstAlg}(n, ta, \alpha, \alpha_1, x_1, p_1),\\ &[\alpha_1, \beta_1, f_1, a_1, h, h_1]\mathsf{AppAbstAlg}(n, ta, \alpha, \alpha_1, \beta_1, f_1, a_1, h, h_1),\\ &\beta,\\ &X)\end{aligned}$$

The combinatorial completeness

$$\begin{aligned}&\mathsf{CombComp}(n, ta, \alpha, \beta, X) =\\ &\vdash_{\mathsf{s}(n),\mathsf{cons}(n,ta,\alpha)} \mathsf{app}(\mathsf{ObInj}(\mathsf{AbstAlg}(X)), \mathsf{var}(\alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha))) \sim_{\mathsf{s}(n),\mathsf{cons}(n,ta,\alpha)} X\end{aligned}$$

where $n : \mathsf{N}, ta : \mathsf{TA}(n), \alpha, \beta : \mathsf{types}, X : \mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta)$. Some arguments were removed from the app,ObInj and AbstAlg.
As with the definition of AbstAlg the term proving this statement is formed with ob_e, so we will prove the premisses of the rule before combining them. The terms have the following typing:

$$\begin{aligned}\mathsf{KCombComp} : \quad &(n : \mathsf{N}; ta : \mathsf{TA}(n); \alpha : \mathsf{types}; \beta_1 : \mathsf{types}; \beta_2 : \mathsf{types})\\ &\mathsf{CombComp}(n, ta, \alpha, \beta_1 \to \beta_2 \to \beta_1, \mathsf{K}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2))\end{aligned}$$

$$\begin{aligned}\mathsf{SCombComp} : \quad &(n : \mathsf{N}; ta : \mathsf{TA}(n); \alpha : \mathsf{types}; \beta_1 : \mathsf{types}; \beta_2 : \mathsf{types}; \beta_3 : \mathsf{types})\\ &\mathsf{CombComp}(n, ta, \alpha, (\beta_1 \to \beta_2 \to \beta_3) \to (\beta_1 \to \beta_2) \to \beta_1 \to \beta_3,\\ &\qquad\qquad \mathsf{S}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, \beta_3))\end{aligned}$$

$$\begin{aligned}\mathsf{VarCombComp} : \quad &(n : \mathsf{N};\\ &ta : \mathsf{TA}(n);\\ &\alpha : \mathsf{types}; \beta : \mathsf{types};\\ &x : \mathsf{IS}(\mathsf{s}(n));\\ &p : \mathsf{VarType}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), x, \beta))\\ &\mathsf{CombComp}(n, ta, \alpha, \beta, \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta, x, p))\end{aligned}$$

$$
\begin{aligned}
\mathsf{AppCombComp} : (\quad & n : \mathsf{N}; ta : \mathsf{TA}(n); \alpha : \mathsf{types}; \beta_1 : \mathsf{types}; \beta_2 : \mathsf{types};\\
& f : \mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1 \to \beta_2);\\
& a : \mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1);\\
& fh : \mathsf{CombComp}(n, ta, \alpha, \beta_1 \to \beta_2, f);\\
& ah : \mathsf{CombComp}(n, ta, \alpha, \beta_1, a))\\
& \mathsf{CombComp}(n, ta, \alpha, \beta_2, \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, f, a))
\end{aligned}
$$

The corresponding terms are:

$$
\begin{aligned}
\mathsf{KCombComp} = [n, ta, \alpha, \beta_1, \beta_2]\mathsf{Keq}(&\mathsf{s}(n),\\
& \mathsf{cons}(n, ta, \alpha),\\
& \beta_1 \to \beta_2 \to \beta_1,\\
& \alpha,\\
& \mathsf{K}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2),\\
& \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha)))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{SCombComp} = [n, ta, \alpha, \beta_1, \beta_2, \beta_3]\mathsf{Keq}(&\mathsf{s}(n),\\
& \mathsf{cons}(n, ta, \alpha),\\
& (\beta_1 \to \beta_2 \to \beta_3) \to (\beta_1 \to \beta_2) \to \beta_1 \to \beta_3,\\
& \alpha,\\
& \mathsf{S}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, \beta_3),\\
& \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha)))
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{VarCombComp} = & [n, ta, \alpha, \beta, x, p]\\
& \mathsf{VarType}_e(n,\\
& \qquad ta,\\
& \qquad [x_1, \beta_1, \alpha_1, p_1]\mathsf{CombComp}(n, ta, \alpha_1, \beta_1,\\
& \qquad\qquad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha_1), \beta_1, x_1, p_1)),\\
& \qquad [\alpha_1]\mathsf{IdEq}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha_1), \alpha_1,\\
& \qquad\qquad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha_1), \alpha_1, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha_1))),\\
& \qquad [\alpha_1, \beta_1, y_1, p_1]\\
& \qquad\quad \mathsf{Keq}(\mathsf{s}(n), \mathsf{cons}(n, ta, \beta_1), \alpha_1, \beta_1,\\
& \qquad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \beta_1), \alpha_1,\\
& \qquad\qquad \mathsf{inject}(n, y), \mathsf{InjectType}(n, ta, \alpha_1, \beta_1, y_1, p_1)),\\
& \qquad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \beta_1), \beta_1, \mathsf{put}(n), \mathsf{PutType}(n, ta, \beta_1))),\\
& \qquad x,\\
& \qquad \beta,\\
& \qquad \alpha,\\
& \qquad p)
\end{aligned}
$$

The case for application requires some definitions that make it more readable:

$$
\begin{aligned}
&\mathsf{fAbst}(n, ta, \alpha, \beta_1, \beta_2, f) : \mathsf{ob}(n, ta, \alpha \to \beta_1 \to \beta_2)\\
&\mathsf{fAbst}(n, ta, \alpha, \beta_1, \beta_2, f) = \mathsf{AbstAlg}(n, ta, \alpha, \beta_1 \to \beta_2, f)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{aAbst}(n, ta, \alpha, \beta, a) : \mathsf{ob}(n, ta, \alpha \to \beta)\\
&\mathsf{aAbst}(n, ta, \alpha, \beta, a) = \mathsf{AbstAlg}(n, ta, \alpha, \beta, a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{fAbstInj}(n, ta, \alpha, \beta_1, \beta_2, f) : \mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha \to \beta_1 \to \beta_2)\\
&\mathsf{fAbstInj}(n, ta, \alpha, \beta_1, \beta_2, f) = \mathsf{ObInj}(n, ta, \alpha \to \beta_1 \to \beta_2, \alpha, \mathsf{fAbst}(n, ta, \alpha, \beta_1, \beta_2, f))
\end{aligned}
$$

$\mathsf{aAbstInj}(n, ta, \alpha, \beta, a) : \mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha \to \beta)$
$\mathsf{aAbstInj}(n, ta, \alpha, \beta, a) = \mathsf{ObInj}(n, ta, \alpha \to \beta, \alpha, \mathsf{aAbst}(n, ta, \alpha, \beta, a))$

$$
\begin{array}{l}
\mathsf{AppCombComp} = [n, ta, \alpha, \beta_1, \beta_2, f, a, fh, ah] \\
\quad \tau(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_2, \\
\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \beta_2, \\
\quad\quad\quad \mathsf{ObInj}(n, ta, \alpha \to \beta_2, \alpha, \\
\quad\quad\quad\quad \mathsf{AbstAlg}(n, ta, \alpha, \beta_2, \\
\quad\quad\quad\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, f, a))), \\
\quad\quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha))), \\
\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, \\
\quad\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \beta_1 \to \beta_2, \\
\quad\quad\quad\quad \mathsf{fAbstInj}, \\
\quad\quad\quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha))), \\
\quad\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \beta_1, \\
\quad\quad\quad\quad \mathsf{aAbstInj}, \\
\quad\quad\quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha)))), \\
\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, f, a), \\
\quad\quad \mathsf{Seq}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \beta_1, \beta_2, \\
\quad\quad\quad \mathsf{fAbstInj}, \\
\quad\quad\quad \mathsf{aAbstInj}, \\
\quad\quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha))), \\
\quad\quad \mu\nu(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta_1, \beta_2, \\
\quad\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \beta_1 \to \beta_2, \\
\quad\quad\quad\quad \mathsf{fAbstInj}, \\
\quad\quad\quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha))), \\
\quad\quad\quad f, \\
\quad\quad\quad \mathsf{app}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \beta_1, \\
\quad\quad\quad\quad \mathsf{aAbstInj}, \\
\quad\quad\quad\quad \mathsf{var}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \alpha, \mathsf{put}(n), \mathsf{PutType}(n, ta, \alpha))), \\
\quad\quad\quad a, \\
\quad\quad\quad fh, \\
\quad\quad\quad ah))
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ObCC} : (n : \mathsf{N})(ta : \mathsf{TA}(n))(\alpha : \mathsf{types})(\beta : \mathsf{types}) \\
\quad (X : \mathsf{ob}(\mathsf{s}(n), \mathsf{cons}(n, ta, \alpha), \beta)) \\
\quad \mathsf{CombComp}(n, ta, \alpha, \beta, X)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ObCC} = [n, ta, \alpha, \beta, X]\mathsf{ob}_e(\mathsf{s}(n), \\
\quad \mathsf{cons}(n, ta, \alpha), \\
\quad [\beta_1, X_1]\mathsf{CombComp}(n, ta, \alpha, \beta_1, X_1), \\
\quad [\alpha_1, \beta_1]\mathsf{KCombComp}(n, ta, \alpha, \alpha_1, \beta_1), \\
\quad [\alpha_1, \beta_1, \gamma_1]\mathsf{SCombComp}(n, ta, \alpha, \alpha_1, \beta_1, \gamma_1), \\
\quad [\alpha_1, x, p]\mathsf{VarCombComp}(n, ta, \alpha, \alpha_1, x, p), \\
\quad [\alpha_1, \beta_1, f, a, h, h_1]\mathsf{AppCombComp}(n, ta, \alpha, \alpha_1, \beta_1, f, a, h, h_1), \\
\quad \beta, \\
\quad X)
\end{array}
$$

Machine Checked Normalization Proofs for Typed Combinator Calculi

draft

Veronica Gaspes*
Jan M. Smith†
University of Göteborg and Chalmers University of Technology

June 1992

1 Introduction

In this paper we will present formalized normalization proofs of two calculi: simply typed combinators and a combinator formulation of Gödel's T. The proofs are based on Tait's computability method [10] and are formalized in Martin-Löf's set theory [6, NPS90]. The motivation for doing these formalizations is to obtain machine checked normalization proofs. The proofs in this paper have been checked using the Alf-system [ACN90]; in fact, given proofs on the level of formalization presented here, it is a simple and straightforward task to have them verified in the set theory implementation in Alf.

The first theory we will discuss, the simply typed combinators, is chosen because of its simplicity: we want a machine checked proof which avoids as much as possible syntactical problems and concentrates on the computability method, which has become a standard tool in normalization proofs for typed theories.

Tait [10] treated a combinator formulation of Gödel's system T [4]. Although that is a much more powerful theory than the simply typed combinators, the normalization proof for Gödel's T is, from the formal point of view, a straightforward extension of that for the simply typed combinators.

The proofs are done in Martin-Löf's set theory with the possibility of introducing families of sets by induction. All the inductive definitions we use are strictly positive and given the introduction rules for such an inductively defined family, an elimination rule, expressing a structural induction rule over the set introduced, can be obtained mechanically [2, 3]. However, the implementation of Martin-Löf's set theory that we have used neither checks the strictly positiveness of the introduction rules, nor the correctness of the elimination rules.

*vero@cs.chalmers.se
†smith@cs.chalmers.se

2 The Simply Typed Combinators

The combinator calculus we will use is the one obtained by the Curry-Howard interpration of positive implicational calculus, formulated in the Hilbert style with the axioms

$$A \supset B \supset A$$

$$(A \supset B \supset C) \supset (A \supset B) \supset A \supset C$$

and with modus ponens as the only derivation rule. So the types will be built up from type variables and the function arrow:

- $X,\ Y,\ Z,\ \ldots$ are types.
- If α and β are types, then $\alpha \to \beta$ is a type.

Corresponding to the axioms, we have, for arbitrary types α, β and γ, constants $\mathbf{K}_{\alpha,\beta}$ and $\mathbf{S}_{\alpha,\beta,\gamma}$ with typings

$$\mathbf{K}_{\alpha,\beta} : \alpha \to \beta \to \alpha$$

$$\mathbf{S}_{\alpha,\beta,\gamma} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$

Modus ponens corresponds to application:

$$\frac{M : \alpha \to \beta \qquad N : \alpha}{MN : \beta}$$

We have the usual contraction rules, in which we assume that all terms are of appropriate types,

$$\mathbf{K}_{\alpha,\beta}\, MN \succ M$$

$$\mathbf{S}_{\alpha,\beta,\gamma}\, MNO \succ (MO)(NO)$$

We will normalize a term by head reduction. So we add the rule

$$\frac{M \succ N}{MO \succ NO}$$

and define inductively what it means for a term to be normalizable:

- $\mathbf{K}_{\alpha,\beta}$ is normalizable.
- $\mathbf{S}_{\alpha,\beta,\gamma}$ is normalizable.
- $\mathbf{K}_{\alpha,\beta}\, M$ is normalizable provided M is normalizable.
- $\mathbf{S}_{\alpha,\beta,\gamma}\, M$ is normalizable provided M is normalizable.
- $\mathbf{S}_{\alpha,\beta,\gamma}\, MN$ is normalizable provided M and N are normalizable.
- If $M \succ N$ and N is normalizable, then M is normalizable.

Intuitively, it is clear that if a term is normalizable then it reduces to a term on normal form, that is, to a term which does not contain any subterm of the form $\mathbf{K}_{\alpha,\beta}\, MN$ or $\mathbf{S}_{\alpha,\beta,\gamma}\, MNO$. This can be rigorously proved by defining the reduction $\succ^*$ as the transitive and symmetric closure of the relation obtained by adding to the above conversion rules

$$\frac{M \succ^\star N}{\mathbf{K}_{\alpha,\beta}\, M \succ^\star \mathbf{K}_{\alpha,\beta}\, N} \quad \frac{M \succ^\star N}{\mathbf{S}_{\alpha,\beta,\gamma}\, M \succ^\star \mathbf{S}_{\alpha,\beta,\gamma}\, N} \quad \frac{M \succ^\star N}{\mathbf{S}_{\alpha,\beta,\gamma}\, MO \succ^\star \mathbf{S}_{\alpha,\beta,\gamma}\, NO} \quad \frac{M \succ^\star N}{\mathbf{S}_{\alpha,\beta,\gamma}\, OM \succ^\star \mathbf{S}_{\alpha,\beta,\gamma}\, ON}$$

and by defining the normal forms inductively by

- $\mathbf{K}_{\alpha,\beta}$ is on normal form.
- $\mathbf{S}_{\alpha,\beta,\gamma}$ is on normal form.
- $\mathbf{K}_{\alpha,\beta}\, M$ is on normal form provided M is on normal form.
- $\mathbf{S}_{\alpha,\beta,\gamma}\, M$ is on normal form provided M is on normal form.
- $\mathbf{S}_{\alpha,\beta,\gamma}\, MN$ is on normal form provided M and N are on normal form.

The idea of defining normalizability directly by induction is due to Martin-Löf [5].

2.1 Normalization Theorem for the Simply Typed Combinators

The proof of the normalization theorem, that is, that every typable term is normalizable, is by Tait's computability method. The idea of this method is that the normalizability of a term M of type α is proved by induction on the derivation of $M : \alpha$ using a stronger induction hypothesis, namely that M is a computable term of type α. For the simply typed combinators the set of computable terms of a type is inductively defined over the types by

- There are no computable terms of a type variable.
- A term $M : \alpha \to \beta$ is computable if
  - M is normalizable and
  - MN is computable of type β provided N is computable of type α.

Note that, from this definition it follows that a computable term is normalizable. Hence, the normalizability theorem is an immediate corollary to the following theorem.

Theorem 1 *If $M : \alpha$ then M is computable of type α.*

In the proof we need the following lemma, which is easily proved by induction on the type α.

Lemma 1 *Let M and N be of type α. If $M \succ N$ and N is computable of type α, then M is computable of type α.*

The proof of the theorem is by induction on the derivation of $M : \alpha$. We then have the following cases.

- $\mathbf{K}_{\alpha,\beta} : \alpha \to \beta \to \alpha$.
  Since $\mathbf{K}_{\alpha,\beta}$ is a term of an arrow type we have to prove, by the definition of computability, that (i) $\mathbf{K}_{\alpha,\beta}$ is normalizable and that (ii) $\mathbf{K}_{\alpha,\beta}\, M$ is computable of type $\beta \to \alpha$ for all computable M of type α. (i) follows from the definition of being normalizable. For the proof of (ii) let M be a computable term of type α. We have to show that $\mathbf{K}_{\alpha,\beta}\, M$ is normalizable and that $\mathbf{K}_{\alpha,\beta}\, MN$ is computable of type α for all computable N of type β. Since M is computable, M is normalizable, hence, by definition, $\mathbf{K}_{\alpha,\beta}\, M$ is normalizable. Since $\mathbf{K}_{\alpha,\beta}\, MN \succ M$ and M is assumed to be computable, lemma 1 gives that $\mathbf{K}_{\alpha,\beta}\, MN$ is computable.

- $\mathbf{S}_{\alpha,\beta,\gamma} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$.
  The argument is similar to that for $\mathbf{K}_{\alpha,\beta}$. Let M, N and O be arbitrary computable terms of type $\alpha \to \beta \to \gamma$, $\alpha \to \beta$ and α, respectively. The definition of computability gives that MO and NO are computable terms of types $\beta \to \gamma$ and β, respectively. By the definition of computability $MO(NO)$ is computable of type γ; since $\mathbf{S}_{\alpha,\beta,\gamma}MNO \succ MO(NO)$, lemma 1 gives that $\mathbf{S}_{\alpha,\beta,\gamma}$ is computable.

- Application

$$\frac{M : \alpha \to \beta \qquad N : \alpha}{(MN) : \beta}$$

  By the induction hypothesis M is computable of type $\alpha \to \beta$ and N is computable of type α. The definition of computability for terms of type $\alpha \to \beta$ then gives that MN is computable of type β.

2.2 Formalization of the normalization proof

The formalization of the normalization proof in Martin-Löf's set theory requires representations of the typed combinators, the relation of contraction of a head redex and of the predicates of normalizability and computability. All these are defined as sets, either by introducing a new set by an inductive definition or, in the case of the computability predicate, by recursion on the type structure of the combinators, using the universe of Martin-Löf's set theory.

The simply typed combinators are introduced by a set Type for the types and a family Term of sets over Type such that the elements in $\mathsf{Term}(\alpha)$ are the terms of type α. The contraction of the head redex is expressed by a family of sets $\mathsf{OneStepHead}(\alpha, M, N)$ where M and N are terms of type α such that N is obtained by contracting the head redex of M.

We follow the notation in [NPS90].

2.2.1 The set of types

We first declare Type to be a set.

Type-formation

$$\mathsf{Type} : \mathit{Set}$$

The set Type is inductively generated and there are two introduction rules; one for the atomic sets, that is, the type variables, and one for the function types. We let the type variables be indexed by some given set Atom which could, for instance, be the set of natural numbers.

Type-introduction 1

$$\frac{a : \mathsf{Atom}}{\mathsf{var}(a) : \mathsf{Type}}$$

Type-introduction 2

$$\frac{\alpha : \mathsf{Type} \qquad \beta : \mathsf{Type}}{\alpha \to \beta : \mathsf{Type}}$$

The elimination rule expresses structural induction on the set Type.

Type-elimination

$$\frac{\begin{array}{l} P(u)\ \mathit{Set}\ [u : \mathsf{Type}] \\ b : P(\mathsf{var}(a))\ [a : \mathsf{Atom}] \\ i(\alpha, \beta, I_\alpha, I_\beta) : P(\alpha \to \beta)\ [\alpha, \beta : \mathsf{Type}, I_\alpha : P(\alpha), I_\beta : P(\beta)] \\ \alpha : \mathsf{Type} \end{array}}{\mathsf{TypeRec}(P, b, i, \alpha) : P(\alpha)}$$

A step in the informal proof by induction on the types will be formalized by an application of Type-elimination.

2.2.2 The family of sets of terms

Term is declared to be a family of sets over the types.

Term-formation

$$\mathsf{Term}(\alpha) : \mathit{Set}\ [\alpha : \mathsf{Type}]$$

The family is defined inductively by introduction rules which correspond to the formation of the typed combinators.

Term-introduction 1

$$\frac{\alpha, \beta : \mathsf{Type}}{\mathsf{K}(\alpha, \beta) : \mathsf{Term}(\alpha \to \beta)}$$

Term-introduction 2

$$\frac{\alpha, \beta, \gamma : \mathsf{Type}}{\mathsf{S}(\alpha, \beta, \gamma) : \mathsf{Term}((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma)}$$

Term-introduction 3

$$\frac{\begin{array}{l} \alpha, \beta : \mathsf{Type} \\ M : \mathsf{Term}(\alpha \to \beta) \\ N : \mathsf{Term}(\alpha) \end{array}}{\mathsf{App}(\alpha, \beta, M, N) : \mathsf{Term}(\beta)}$$

In the sequel we will often omit the type arguments in the constructor App since they can be obtained from the types of the other arguments. The elimination rule completes the definition by expressing structural induction on the family Term.

Term-elimination

$$\frac{\begin{array}{l} P(u, v) : \mathit{Set}\ [u : \mathsf{Type}, v : \mathsf{Term}(u)] \\ k(x, y) : P(x \to y \to z, \mathsf{K}(x, y))\ [x, y : \mathsf{Type}] \\ s(x, y, z) : P((x \to y \to z) \to (x \to y) \to x \to z, \mathsf{S}(x, y, z))\ [x, y, z : \mathsf{Type}] \\ ap(x, y, u, v, I_u, I_v) : P(y, \mathsf{App}(u, v)) \left[\begin{array}{l} x, y : \mathsf{Type}, \\ u : \mathsf{Term}(x \to y), \\ v : \mathsf{Term}(x) \\ I_u : P(x \to y, u), \\ I_v : P(x, v) \end{array} \right] \\ \alpha : \mathsf{Type} \\ M : \mathsf{Term}(\alpha) \end{array}}{\mathsf{TermRec}(P, k, s, ap, \alpha, M) : P(\alpha, M)}$$

When informally a statement is proved by induction on the derivation of $t : \alpha$, the formal proof is by an application of Term-elimination.

2.2.3 The family of sets expressing head contraction

$\mathsf{OneStepHead}$ is declared to be a family of sets over types and terms.

$\mathsf{OneStepHead}$-formation

$$\mathsf{OneStepHead}(\alpha, M, N) : \mathit{Set}\ [\alpha : \mathsf{Type},\ M, N : \mathsf{Term}(\alpha)]$$

The set $\mathsf{OneStepHead}(\alpha, M, N)$ expresses that N results from contracting the head redex of M; thus it is defined by the following introduction rules.

$\mathsf{OneStepHead}$-introduction 1

$$\frac{\alpha, \beta : \mathsf{Type} \quad M : \mathsf{Term}(\alpha) \quad M : \mathsf{Term}(\beta)}{\mathsf{kk}(\alpha, \beta, M, N) : \mathsf{OneStepHead}(\alpha, \mathsf{App}(\mathsf{App}(\mathsf{K}(\alpha, \beta), M), N), M)}$$

$\mathsf{OneStepHead}$-introduction 2

$$\frac{\alpha, \beta, \gamma : \mathsf{Type} \quad M : \mathsf{Term}(\alpha \to \beta \to \gamma) \quad N : \mathsf{Term}(\alpha \to \beta) \quad O : \mathsf{Term}(\alpha)}{\begin{array}{l}\mathsf{ss}(\alpha, \beta, M, N, O) : \\ \qquad \mathsf{OneStepHead}(\gamma, \mathsf{App}(\mathsf{App}(\mathsf{App}(\mathsf{S}(\alpha, \beta, \gamma), M), N), O), \mathsf{App}(\mathsf{App}(M, O), \mathsf{App}(N, O)))\end{array}}$$

$\mathsf{OneStepHead}$-introduction 3

$$\frac{\alpha, \beta : \mathsf{Type} \quad M, N : \mathsf{Term}(\alpha \to \beta) \quad O : \mathsf{Term}(\alpha) \quad p : \mathsf{OneStepHead}(\alpha \to \beta, M, N)}{\mu(\alpha, \beta, M, N, O, p) : \mathsf{OneStepHead}(\beta, \mathsf{App}(M, O), \mathsf{App}(N, O))}$$

The elimination rule for this inductive definition of $\mathsf{OneStepHead}$ will not be used in the normalization proof and we refrain from formulating it.

2.2.4 The family of sets formalizing normalizability

The family Norm is introduced for the proofs that a term of a given type is normalizable.

Norm-formation

$$\mathsf{Norm}(\alpha, M) : \mathit{Set}\ [\alpha : \mathsf{Type},\ M : \mathsf{Term}(\alpha)]$$

The introduction rules correspond to the clauses in the inductive definition of the set of normalizable terms of a given type.

Norm-introduction 1

$$\frac{\alpha, \beta : \mathsf{Type}}{\mathsf{kn}(\alpha, \beta) : \mathsf{Norm}(\alpha \to \beta \to \alpha, \mathsf{K}(\alpha, \beta))}$$

Norm-introduction 2

$$\frac{\alpha, \beta, \gamma : \mathsf{Type}}{\mathsf{sn}(\alpha, \beta, \gamma) : \mathsf{Norm}((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathsf{S}(\alpha, \beta, \gamma))}$$

Norm-introduction 3

$$\frac{\alpha, \beta : \mathsf{Type} \quad M : \mathsf{Term}(\alpha) \quad M_n : \mathsf{Norm}(\alpha, M)}{\mathsf{kan}(\alpha, \beta, M, M_n) : \mathsf{Norm}(\beta \to \alpha, \mathsf{App}(\mathsf{K}, M))}$$

Norm-introduction 4

$$\frac{\alpha, \beta, \gamma : \mathsf{Type} \quad M : \mathsf{Term}(\alpha \to \beta \to \gamma) \quad M_n : \mathsf{Norm}(\alpha \to \beta \to \gamma, M)}{\mathsf{sfn}(\alpha, \beta, \gamma, M, M_n) : \mathsf{Norm}((\alpha \to \beta) \to \alpha \to \gamma, \mathsf{App}(\mathsf{S}, M))}$$

Norm-introduction 5

$$\frac{\begin{array}{l} \alpha, \beta, \gamma : \mathsf{Type} \\ M : \mathsf{Term}(\alpha \to \beta \to \gamma) \\ N : \mathsf{Term}(\alpha \to \beta) \\ M_n : \mathsf{Norm}(\alpha \to \beta \to \gamma, M) \\ N_n : \mathsf{Norm}(\alpha \to \beta, N) \end{array}}{\mathsf{sfgn}(\alpha, \beta, \gamma, M, N, M_n, N_n) : \mathsf{Norm}(\alpha \to \gamma, \mathsf{App}(\mathsf{App}(\mathsf{S}, M), N))}$$

Norm-introduction 6

$$\frac{\alpha : \mathsf{Type} \quad M, N : \mathsf{Term}(\alpha) \quad p : \mathsf{OneStepHead}(\alpha, M, N) \quad N_n : \mathsf{Norm}(\alpha, N)}{\mathsf{mrn}(\alpha, M, N, p, N_n) : \mathsf{Norm}(\alpha, M)}$$

The elimination rule for Norm will not be used in the proof.

2.2.5 The family of sets for the computability predicate

The computability predicate

$$\mathsf{Comp}(\alpha, M) : Set\ [\alpha : \mathsf{Type}, M : \mathsf{Term}(\alpha)]$$

should satisfy the equations

$$\mathsf{Comp}(\mathsf{var}(a), M) \ = \bot \ \ [a : \mathsf{Atom}, \ M : \mathsf{Term}(\mathsf{var}(a))]$$

$$\mathsf{Comp}(\alpha \to \beta, M) \ = \ \mathsf{Norm}(\alpha \to \beta, M) \wedge (\Pi N : \mathsf{Term}(\alpha))\mathsf{Comp}(\alpha, N) \supset \mathsf{Comp}(\beta, \mathsf{App}(M, N))$$

The definition of Comp is by recursion on the type structure either by using a universe or directly by set valued recursion [9]. Using the notation in [NPS90] the definition of Comp using a universe would be

$$\mathsf{Comp}(\alpha) \ = \ \mathsf{Set}(\widehat{\mathsf{Comp}}(\alpha))$$

where

$$\widehat{\mathsf{Comp}}(\alpha) \ = \ \mathsf{TypeRec}([v]\mathsf{Term}(v) \Rightarrow \mathsf{U}, [at]BaseCase(at), [\gamma, \delta, I_\gamma, I_\delta]ArrowCase(\gamma, \delta, I_\gamma, I_\delta))$$

$$BaseCase(at) \equiv \lambda M.\widehat{\bot}$$

$$ArrowCase(\gamma, \delta, I_\gamma, I_\delta) \equiv \lambda M.\widehat{\mathsf{Norm}}(M) \widehat{\wedge} (\widehat{\Pi} N \in \widehat{\mathsf{Term}}(\gamma))(I_\gamma(N) \widehat{\Rightarrow} I_\delta(\mathsf{App}(M, N)))$$

The definition of the computability predicate using the approach in [9] would be the same except that no coding would be involved. In fact, in the implementation of the proof in Alf, we wrote the two defining equations for Comp directly, using pattern matching; in fact this approach makes the equations for the computability predicate definitional.

2.2.6 The formal normalization proof

The proposition expressing that every typable term is normalizable can now be formulated in Martin-Löf's set theory by

$$(\Pi\alpha : \mathsf{Type})(\Pi t : \mathsf{Term}(\alpha))\mathsf{Norm}(\alpha, t)$$

The formalization of the proof of the normalization theorem consists in deriving an element in this set. Such an element is built up by applying a proof of

$$(\Pi\alpha : \mathsf{Type})(\Pi t : \mathsf{Term}(\alpha))(\mathsf{Comp}(\alpha, t) \supset \mathsf{Norm}(\alpha, t))$$

to a proof of

$$(\Pi\alpha : \mathsf{Type})(\Pi t : \mathsf{Term}(\alpha))\mathsf{Comp}(\alpha, t)$$

The first of these statements, which in the informal presentation of the proof was assumed to be a direct consequence of the definition of computability, requires an explicit proof in the formalization; the proof is by induction on the type α, i.e. by Type-elimination.

The second statement is proved by induction on the derivation of $t : \mathsf{Term}(\alpha)$, i.e. by Term-elimination, using the lemmata

$$(\Pi\alpha : \mathsf{Type})(\Pi t, u : \mathsf{Term}(\alpha))\mathsf{OneStepHead}(\alpha, t, u) \supset \mathsf{Comp}(\alpha, u) \supset \mathsf{Comp}(\alpha, t)$$

and

$$\begin{gathered}(\Pi\alpha, \beta : \mathsf{Type})(\Pi f \in \mathsf{Term}(\alpha \to \beta))(\Pi a : \mathsf{Term}(\alpha)) \\ \mathsf{Comp}(\alpha \to \beta, f) \supset \mathsf{Comp}(\alpha, a) \supset \mathsf{Comp}(\beta, \mathsf{App}(f, a))\end{gathered}$$

The first lemma was proved by induction on α and the second is an immediate consequence of the definition of $\mathsf{Comp}(\alpha \to \beta, f)$.

Instead of presenting the proofs in the tree like fashion that results from the natural deduction style used above when formulating the inference rules, we will express these rules in the logical framework as functions. For example, the introduction rule for normalizability

$$\frac{\alpha : \mathsf{Type} \quad M, N : \mathsf{Term}(\alpha) \quad p : \mathsf{OneStepHead}(\alpha, M, N) \quad N_n : \mathsf{Norm}(\alpha, N)}{\mathsf{mrn}(\alpha, M, N, p, N_n) : \mathsf{Norm}(\alpha, M)}$$

is expressed by the function

$$\mathsf{mrn} : (\alpha : \mathsf{Type})(M, N : \mathsf{Term}(\alpha))(p : \mathsf{OneStepHead}(\alpha, M, N))(N_n : \mathsf{Norm}(\alpha, N))\mathsf{Norm}(\alpha, M)$$

We refer to [NPS90] for expressing rules in the logical framework.

Formally, the normalization theorem is proved by the term

$$\mathsf{AllNorm} : (\alpha : \mathsf{Type})(t : \mathsf{Term}(\alpha))\mathsf{Norm}(\alpha, t)$$

$$\begin{aligned}\mathsf{AllNorm}(\alpha, t) = \mathsf{apply}(&\mathsf{Comp}(\alpha, t),\\ &\mathsf{Norm}(\alpha, t),\\ &\mathsf{apply}(\mathsf{Term}(\alpha),\\ &\qquad [a]\mathsf{Comp}(\alpha, a) \supset \mathsf{Norm}(\alpha, a),\\ &\qquad \mathsf{lemma}(\alpha),\\ &\qquad t),\\ &\mathsf{AllComp}(\alpha, t))\end{aligned}$$

Removing all set parameters, this term becomes

$$\mathsf{AllNorm}(\alpha, t) = \mathsf{apply}(\mathsf{apply}(\mathsf{lemma}(\alpha), t), \mathsf{AllComp}(\alpha, t))$$

We will often use this convention of hiding the set parameters of the constants.

The term AllComp proves that every term is computable (by induction on the definition of being a term in a given type) and the term lemma proves that every computable term is normalizable (by induction on the types):

$$\mathsf{lemma} : (\alpha : \mathsf{Type})\Pi(\mathsf{Term}(\alpha), [t]\mathsf{Comp}(\alpha, t) \supset \mathsf{Norm}(\alpha, t))$$

$$\begin{aligned}\mathsf{lemma}(\alpha) = \mathsf{TypeRec}(&[a]\Pi(\mathsf{Term}(a), [t]\mathsf{Comp}(a, t) \supset \mathsf{Norm}(a, t)),\\ &[at]\lambda([x]\lambda([cx]\perp_e([v]\mathsf{Norm}(\mathsf{var}(at), x), cx))),\\ &[\beta, \gamma, h_\beta, h_\gamma]\lambda([f]\lambda([cf]\mathsf{fst}(cf))),\\ &\alpha)\end{aligned}$$

$$\mathsf{AllComp} : (\alpha : \mathsf{Type})(t : \mathsf{Term}(\alpha))\mathsf{Comp}(\alpha, t)$$

$$\begin{aligned}\mathsf{AllComp}(\alpha, t) = \mathsf{TermRec}(&[\gamma, u]\mathsf{Comp}(\gamma, u),\\ &[\beta_1, \beta_2]\mathsf{Kcomp}(\beta_1, \beta_2),\\ &[\beta_1, \beta_2, \beta_3]\mathsf{Scomp}(\beta_1, \beta_2, \beta_3),\\ &[\beta_1, \beta_2, f, b, h_f, h_b]\mathsf{Appcomp}(\beta_1, \beta_2, f, b, h_f, h_b),\\ &\alpha,\\ &t)\end{aligned}$$

The definition of AllComp requires the definitions of Kcomp, Scomp and Appcomp which, respectively, prove that K, S, and $\mathsf{App}(f, a)$ are computable. We will here only give the definition of Appcomp; the definitions of the other two are somewhat longer, reflecting that the proofs of the computability of K and S are slightly more complicated.

$$\begin{aligned}\mathsf{Appcomp} \;:\; &(\alpha, \beta : \mathsf{Type})\\ &(f : \mathsf{Term}(\alpha \to \beta))\\ &(a : \mathsf{Term}(\alpha))\\ &(cf : \mathsf{Comp}(\alpha \to \beta, f))\\ &(ca : \mathsf{Comp}(\alpha, a))\\ &\mathsf{Comp}(\beta, \mathsf{App}(\alpha, \beta, f, a))\end{aligned}$$

$$\mathsf{Appcomp} = [\alpha, \beta, f, a, cf, ca]\mathsf{apply}(\mathsf{apply}(\mathsf{snd}(cf), a), ca)$$

2.2.7 An example

As an illustration of the computational content of the formal proof, we will show the proof object for the normalizability of the combinator $\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$, where I_α is the identity function on the type α and ι is an atomic type. Omitting the type information in the combinators $\mathbf{K}_{\alpha,\beta}$ and $\mathbf{S}_{\alpha,\beta,\gamma}$ the identity function is defined by

$$\mathsf{I}_\alpha = \mathbf{SKK}$$

and the head reduction to normal form of $\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$ is

$$\begin{gathered}\mathsf{SKK}((\mathsf{SKK})(\mathsf{SKK})) \succ (\mathsf{K}((\mathsf{SKK})(\mathsf{SKK})))(\mathsf{K}((\mathsf{SKK})(\mathsf{SKK}))) \succ\\ (\mathsf{SKK})(\mathsf{SKK}) \succ (\mathsf{K}(\mathsf{SKK}))(\mathsf{K}(\mathsf{SKK})) \succ \mathsf{SKK} = \mathsf{I}_\iota\end{gathered}$$

This reduction sequence will appear in the proof term.

For the formal proof we need an element $i : \mathsf{Atom}$ and the define ι to be $\mathsf{var}(i)$. The combinators I_ι, $\mathsf{I}_{\iota\to\iota}$ and $\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$ are introduced by the explicit definitions

$$\mathsf{I}_\iota : \mathsf{Term}(\iota \to \iota)$$

$$\mathsf{I}_\iota = \mathsf{App}(\mathsf{App}(\mathsf{S}(\iota, \iota \to \iota, \iota), \mathsf{K}(\iota, \iota \to \iota)), \mathsf{K}(\iota, \iota)),$$

$$\mathsf{I}_{\iota\to\iota} : \mathsf{Term}((\iota \to \iota) \to \iota \to \iota)$$

$$\begin{aligned}\mathsf{I}_{\iota\to\iota} = \mathsf{App}(\mathsf{App}(&\mathsf{S}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota, \iota \to \iota),\\ &\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota)),\\ &\mathsf{K}(\iota \to \iota, \iota \to \iota))\end{aligned}$$

and

$$\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota) : \mathsf{Term}(\iota \to \iota)$$

$$\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota) = \mathsf{App}(\iota \to \iota, \iota \to \iota, \mathsf{I}_{\iota\to\iota}, \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$$

respectively. Both in these definitions and in the proof term below many type arguments of the constants have been removed for the sake of readability.

The proof of the normalizability of $\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$ is by showing that the terms to which $\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$ head reduces are normalizable; hence, the proof consists of the series of terms in the reduction together with proofs of these reductions and proofs that the corresponding terms are normalizable.

Substituting $\iota \to \iota$ and $\mathsf{I}_{\iota\to\iota}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$ for α and t, respectively, in $\mathsf{AllNorm}(\alpha, t)$ and unfolding definitions, we obtain

$$\mathsf{AllNorm}(\iota \to \iota, \mathsf{I}_{\iota\to\iota}\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota) : \mathsf{Norm}(\iota \to \iota, \mathsf{I}_{\iota\to\iota}\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)$$

$$\begin{aligned}
&\mathsf{AllNorm}(\iota \to \iota, \mathsf{I}_{\iota\to\iota}\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota) =\\
&\mathsf{mrn}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota,\\
&\quad \mathsf{App}(\mathsf{App}(\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota), \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota), \mathsf{App}(\mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)),\\
&\quad \mathsf{ss}(\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota), \mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota),\\
&\quad \mathsf{mrn}(\mathsf{App}(\mathsf{App}(\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota), \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota), \mathsf{App}(\mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)),\\
&\qquad \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota,\\
&\qquad \mathsf{kk}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota, \mathsf{App}(\mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota)),\\
&\qquad \mathsf{mrn}(\mathsf{I}_{\iota\to\iota}\mathsf{I}_\iota,\\
&\qquad\quad \mathsf{App}(\mathsf{App}(\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota), \mathsf{I}_\iota), \mathsf{App}(\mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_\iota)),\\
&\qquad\quad \mathsf{ss}(\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota), \mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_\iota),\\
&\qquad\quad \mathsf{mrn}(\mathsf{App}(\mathsf{App}(\mathsf{K}(\iota \to \iota, (\iota \to \iota) \to \iota \to \iota), \mathsf{I}_\iota), \mathsf{App}(\mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_\iota)),\\
&\qquad\qquad \mathsf{I}_\iota,\\
&\qquad\qquad \mathsf{kk}(\mathsf{I}_\iota, \mathsf{App}(\mathsf{K}(\iota \to \iota, \iota \to \iota), \mathsf{I}_\iota)),\\
&\qquad\qquad \mathsf{sfgn}(\mathsf{K}(\iota, \iota \to \iota), \mathsf{K}(\iota, \iota), \mathsf{kn}, \mathsf{kn})))))
\end{aligned}$$

2.2.8 Extracting the normal form from the proof

In the above example, we have seen that when the proof of the normalizability theorem is applied on a combinator, the normal form appears in the proof. We will now define a function

$$\mathsf{extraction} : (\alpha : \mathsf{Type})(t : \mathsf{Term}(\alpha))(p : \mathsf{Norm}(\alpha, t))\mathsf{Term}(\alpha)$$

that from a proof p of the normalizability of a given combinator extracts the the normal form of the combinator. The function will be defined by induction on the normalizability of the term. The cases that correspond to the definition by elimination of the set Norm applied to p are:

$$\mathsf{extraction}(\alpha \to \beta \to \alpha, \mathsf{K}(\alpha, \beta), \mathsf{kn}(\alpha, \beta)) = \mathsf{K}(\alpha, \beta)$$

$$\mathsf{extraction}((\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathsf{S}(\alpha, \beta, \gamma), \mathsf{sn}(\alpha, \beta, \gamma)) = \mathsf{S}(\alpha, \beta, \gamma)$$

$$\begin{aligned}&\mathsf{extraction}(\beta \to \alpha, \mathsf{App}(\mathsf{K}(\alpha, \beta), a), \mathsf{kan}(\alpha, \beta, a, p)) = \\ &\qquad\qquad\mathsf{App}(\mathsf{K}(\alpha, \beta), \mathsf{extraction}(\alpha, a, p))\end{aligned}$$

$$\begin{aligned}&\mathsf{extraction}((\alpha \to \beta) \to \alpha \to \gamma,\\ &\qquad \mathsf{App}(\mathsf{S}(\alpha, \beta, \gamma), f),\\ &\qquad \mathsf{sfn}(\alpha, \beta, \gamma, f, p))\\ &\qquad =\\ &\qquad \mathsf{App}(\mathsf{S}(\alpha, \beta, \gamma), \mathsf{extraction}(\alpha \to \beta \to \gamma, f, p))\end{aligned}$$

$$\begin{aligned}&\mathsf{extraction}(\alpha \to \gamma,\\ &\qquad \mathsf{App}(\mathsf{App}(\mathsf{S}(\alpha, \beta, \gamma), f), g),\\ &\qquad \mathsf{sfgn}(\alpha, \beta, \gamma, f, g, p, q))\\ &\qquad =\\ &\qquad \mathsf{App}(\mathsf{App}(\mathsf{S}(\alpha, \beta, \gamma), \mathsf{extraction}(\alpha \to \beta \to \gamma, f, p)), \mathsf{extraction}(\alpha \to \beta, g, q))\end{aligned}$$

$$\mathsf{extraction}(\alpha, t, \mathsf{mrn}(\alpha, t, u, p, q)) = \mathsf{extraction}(\alpha, u, q)$$

This extraction function may then be applied to the proof AllNorm of the normalizability of all terms, thus we obtain an algorithm that computes the normal form of a term:

$$\mathsf{nf} = [\alpha, t]\mathsf{extraction}(\alpha, t, \mathsf{AllNorm}(\alpha, t)) : (\alpha : \mathsf{Type})(t : \mathsf{Term}(\alpha))\mathsf{Term}(\alpha)$$

3 Gödel's System T

A combinator formulation Gödel's system T of computable functionals [8, 10] is obtained from the theory of the simply type combinators by replacing the type variables by the set N of natural numbers. The terms are then obtained by deleting the type variables and adding

$$\mathsf{0} : \mathsf{N}$$

$$\mathsf{s} : \mathsf{N} \to \mathsf{N}$$

$$\mathbf{R}_\alpha : \alpha \to (\alpha \to \mathsf{N} \to \alpha) \to \mathsf{N} \to \alpha$$

The contraction relation is extended by the rules of the recursion operator:

$$\mathbf{R}_\alpha M N\, \mathsf{0} \succ M$$

$$\mathbf{R}_\alpha M N\, (\mathsf{s}T) \succ N\, (\mathbf{R}_\alpha M N T)\, T$$

$$\frac{O \succ P}{\mathbf{R}_\alpha MNO \succ \mathbf{R}_\alpha MNP}$$

To the definition of the set of normalizable terms we add the following clauses:

- $\mathsf{0}$ is normalizable.
- s is normalizable.
- $\mathsf{s}M$ is normalizable provided M is normalizable.
- $\mathbf{R}_\alpha$ is normalizable.
- $\mathbf{R}_\alpha\, M$ is normalizable provided M is normalizable.
- $\mathbf{R}_\alpha\, MN$ is normalizable provided M and N are normalizable.

3.1 Normalization Theorem for Gödel's System T

To obtain a proof of the normalizability theorem for Gödel's T from the proof for the simply typed combinators, we have to change the definition of the computability predicate in the atomic case. For the simply typed combinators the atomic case was empty; now it is the set of computable natural numbers:

- $\mathsf{0}$ is computable of type N.
- $\mathsf{s}M$ is computable of type N provided M is computable of type N.
- If $M \succ N$ and N is computable of type N, then M is computable.

Formally, the set of computable natural numbers is introduced by inductively defined family CompN:

CompN Formation

$$\mathsf{CompN}(M)\ Set[M \in \mathsf{Term}(\mathsf{N})]$$

CompN Introduction

$$\mathsf{zComp} \in \mathsf{CompN}(\mathsf{0})$$

$$\frac{M : \mathsf{Term}(\mathsf{N}) \quad p : \mathsf{CompN}(M)}{\mathsf{sComp}(M) : \mathsf{CompN}(\mathsf{s}(M))}$$

$$\frac{M, N : \mathsf{Term}(\mathsf{N}) \quad p : \mathsf{OneStepHead}(M, N, \mathsf{N}) \quad q : \mathsf{CompN}(N)}{\mathsf{mrComp}(M, N, p, q) : \mathsf{CompN}(M)}$$

CompN Elimination

$$\frac{\begin{array}{l} P(u,v) : Set\ [u : \mathsf{Term}(\mathsf{N}), v : \mathsf{CompN}(u)] \\ zz : P(\mathsf{0}, \mathsf{zComp}) \\ ss(x,y,z) : P(\mathsf{s}(x), \mathsf{sComp}(x))\ [x : \mathsf{Term}(\mathsf{N}), y : \mathsf{CompN}(x), z : P(x,y)] \\ mrc(x,y,u,v,I_y) : P(x, \mathsf{mrComp}(x,y,u,v)) \left[\begin{array}{l} x, y : \mathsf{Term}(\mathsf{N}), \\ u : \mathsf{OneStepHead}(x,y,\mathsf{N}), \\ v : \mathsf{CompN}(y) \\ I_y : P(y,v) \end{array}\right] \\ M : \mathsf{Term}(\mathsf{N}) \\ cn : \mathsf{CompN}(n) \end{array}}{\mathsf{CompNRec}(P, zz, ss, mrc, M, cm) : P(n, cm)}$$

Similarly to the case of the simply typed combinators, the computability predicate is defined so that it satisfies the equations

$$\mathsf{Comp}(\mathsf{N}, M) \ = \ \mathsf{CompN}(M)\ [M : \mathsf{Term}(\mathsf{N})]$$

$$\begin{array}{l} \mathsf{Comp}(\alpha \to \beta, M) = \\ \qquad \mathsf{Norm}(\alpha \to \beta, M) \wedge (\Pi N : \mathsf{Term}(\alpha))\mathsf{Comp}(\alpha, N) \supset \mathsf{Comp}(\beta, \mathsf{App}(M, N)) \end{array}$$

To the proof that every simply typed combinator is computable, we must add the cases:

- $\mathsf{0} \in \mathsf{N}$, which is computable outright.
- $\mathsf{s} \in \mathsf{N} \to \mathsf{N}$. Since s is in an arrow type, we must show that (i) s is normalizable, which follows immediately from the definition of normalizability, and that (ii) $\mathsf{s}M$ is computable of type N for an arbitrary computable M of type N, which follows from one of the clauses defining computability of type N.
- $\mathbf{R}_\alpha : \alpha{\to}(\alpha{\to}\mathsf{N}{\to}\alpha){\to}\mathsf{N}{\to}\alpha$. Since $\mathbf{R}_\alpha$ is in a function type we must be show that (i) $\mathbf{R}_\alpha$ is normalizable, which it is by definition of normalizability, and that (ii) $\mathbf{R}_\alpha M$ is computable of type $(\alpha{\to}\mathsf{N}{\to}\alpha){\to}\mathsf{N}{\to}\alpha$ for arbitrary computable M of type α. The proof of (ii) is by taking an arbitrary M computable of type α and showing that (iii) $\mathbf{R}_\alpha M$ is normalizable, which follows from the computability of M, and (iv) $\mathbf{R}_\alpha MN$ is computable in $\mathsf{N}{\to}\alpha$ for arbitrary N computable of type $\alpha{\to}\mathsf{N}{\to}\alpha$. Again, the proof is by taking an arbitrary computable N of type $\alpha{\to}\mathsf{N}{\to}\alpha$ and show that (v) $\mathbf{R}_\alpha MN$ is normalizable, which it is because both M and N are computable, and (vi) $R_\alpha MNO$ is computable of type α for arbitrary O computable of type N. The proof of (vi) is by induction on the computability of type N of O; thus it has to be proved that
  - $\mathbf{R}_\alpha MN\mathsf{0}$ is computable of type α, which follows from $\mathbf{R}_\alpha MN\mathsf{0} \succ M$.
  - $\mathbf{R}_\alpha MN(\mathsf{s}P)$ is computable of type α, which is a consequence of

    $$\mathbf{R}_\alpha MN(\mathsf{s}P) \succ N(\mathbf{R}_\alpha MNP)P$$

    , where the computability of $N(\mathbf{R}_\alpha MNP)P$ follows from the induction hypothesis and the computability of N and P.
  - $\mathbf{R}_\alpha MNP$ is computable of type α where $P \succ Q$ and Q is computable of type N. The induction hypothesis gives that $\mathbf{R}_\alpha MNQ$ is computable of type α; hence $\mathbf{R}_\alpha MNP \succ \mathbf{R}_\alpha MNQ$ gives the result.

Let R be the formal constant expressing the recursion operator; let Rn, Ran and Rain be the constants introduced in the inductive definition of normalizability; let Rzr, Rsr and Rtn be the constants introduced in the inductive definition of one step head reduction. The following is the formal proof that R is computable:

$$\mathsf{RComp} : (\alpha : \mathsf{Type})\mathsf{Comp}(\alpha \to (\alpha \to \mathsf{N} \to \alpha) \to \mathsf{N} \to \alpha, \mathsf{R}(\alpha))$$

$$\begin{array}{l}
\mathsf{RComp}(\alpha) = \\
\mathsf{pair}(\mathsf{Rn}(\alpha), \\
\quad \lambda([a]\lambda([ca]\mathsf{pair}(\mathsf{Ran}(\alpha, a, \mathsf{lemma}_1(\alpha, a, ca)), \\
\qquad\qquad \lambda([i]\lambda([ci]\mathsf{pair}(\mathsf{Rain}(\alpha, a, i, \mathsf{lemma}_1(\alpha, a, ca), \mathsf{lemma}_1(\alpha \to \mathsf{N} \to \alpha, i, ci)), \\
\qquad\qquad\qquad \lambda([n]\lambda([cn]\mathsf{RComp}')
\end{array}$$

where RComp', which is introduced only for the sake of readability, is the term which realizes the induction on cn, i.e. the computability of n:

$$\begin{array}{l}
\mathsf{CompNRec}([t, ct]\mathsf{Comp}(\alpha, \mathsf{App}(\mathsf{App}(\mathsf{App}(\mathsf{R}(\alpha), a), i), t)), \\
\qquad \mathsf{lemma}_2(\alpha, \mathsf{App}(\mathsf{App}(\mathsf{App}(R(\alpha), a), i), 0), a, \mathsf{Rzr}(\alpha, a, i), ca), \\
\qquad [t, ct, h]\mathsf{lemma}_2(\alpha, \\
\qquad\qquad \mathsf{App}(\mathsf{App}(\mathsf{App}(R(\alpha), a), i), \mathsf{App}(\mathsf{s}, t)), \\
\qquad\qquad \mathsf{App}(\mathsf{App}(i, \mathsf{App}(\mathsf{App}(\mathsf{App}(\mathsf{R}(\alpha), a), i), t)), t), \\
\qquad\qquad \mathsf{Rsr}(\alpha, a, i, t), \\
\qquad\qquad \mathsf{lemma}_3(\alpha, a, i, t, ci, ct, h)), \\
\qquad [t, u, mr, ct, h]\mathsf{lemma}_2(\alpha, \\
\qquad\qquad\quad \mathsf{App}(\mathsf{App}(\mathsf{App}(\mathsf{R}(\alpha), a), i), u), \\
\qquad\qquad\quad \mathsf{App}(\mathsf{App}(\mathsf{App}(\mathsf{R}(\alpha), a), i), t), \\
\qquad\qquad\quad \mathsf{Rtn}(\alpha, a, i, t, u, mr), \\
\qquad\qquad\quad h), \\
\qquad n, \\
\qquad cn))))))))))
\end{array}$$

In the proof, the terms $\mathsf{lemma}_1, \mathsf{lemma}_2, \mathsf{lemma}_3$ are proofs, which are the same as for the simply typed calculus, of

$$\mathsf{lemma}_1 : (\alpha : \mathsf{Type})(a : \mathsf{Term}(\alpha))(ca : \mathsf{Comp}(\alpha, a))\mathsf{Norm}(\alpha, a)$$

$$\begin{array}{l}
\mathsf{lemma}_2 : (\alpha : \mathsf{Type}) \\
\qquad (t : \mathsf{Term}(\alpha)) \\
\qquad (u : \mathsf{Term}(\alpha)) \\
\qquad (\mathsf{OneStepHead}(\alpha, t, u)) \\
\qquad (\mathsf{Comp}(\alpha, u)) \\
\qquad \mathsf{Comp}(\alpha, t)
\end{array}$$

$$\begin{array}{l}
\mathsf{lemma}_3 : (\alpha : \mathsf{Type}) \\
\qquad (a : \mathsf{Term}(\alpha)) \\
\qquad (i : \mathsf{Term}(\alpha \to \mathsf{N} \to \alpha)) \\
\qquad (t : \mathsf{Term}(\mathsf{N})) \\
\qquad (ci : \mathsf{Comp}(\alpha \to \mathsf{N} \to \alpha, i)) \\
\qquad (ct : \mathsf{Comp}(\mathsf{N}, t)) \\
\qquad (h : \mathsf{Comp}(\alpha, \mathsf{App}(\mathsf{App}(\mathsf{App}(R(\alpha), a), i), t))) \\
\qquad \mathsf{Comp}(\alpha, \mathsf{App}(\mathsf{App}(i, \mathsf{App}(\mathsf{App}(\mathsf{App}(\mathsf{R}(\alpha), a), i), t)), t))
\end{array}$$

References

[1] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[2] Thierry Coquand and Christine Paulin. Inductively defined types. In *Proceedings of COLOG-88*, number 417 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[3] 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.

[4] Kurt Gödel. Über eine bisher noch nicht benutze erweitrung des finiten standpunktes. *Dialectica*, 12, 1958.

[5] Per Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland Publishing Company, 1971.

[6] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[7] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[8] L. Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, (8):161–174, 1967.

[9] Jan M. Smith. Propositional Functions and Families of Types. *Notre Dame Journal of Formal Logic*, 30(3), 1989.

[10] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

Inductive and Coinductive types with Iteration and Recursion*

Herman Geuvers†
Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1,
6525 ED Nijmegen,
The Netherlands

July 1992

Abstract

We study (extensions of) simply and polymorphically typed lambda calculus from a point of view of how iterative and recursive functions on inductive types are represented. The inductive types can usually be understood as initial algebras in a certain category and then recursion can be defined in terms of iteration. However, in the syntax we often have only weak initiality, which makes the definition of recursion in terms of iteration inefficient or just impossible. We propose a categorical notion of (primitive) recursion which can easily be added as computation rule to a typed lambda calculus and gives us a clear view on what the dual of recursion, corecursion, on coinductive types is. (The same notion has, independently, been proposed by [Mendler 1991].) We look at how these syntactic notions work out in the simply typed lambda calculus and the polymorphic lambda calculus. It will turn out that in the syntax, recursion can be defined in terms of corecursion and vice versa using polymorphism: Polymorphic lambda calculus with a scheme for either recursion or corecursion suffices to be able to define the other. We compare our syntax for recursion and corecursion with that of Mendler ([Mendler 1987]) and use the latter to obtain meta properties as confluence and normalization.

1 Introduction

In this paper we want to look at formalizations of inductive and coinductive types in different typed lambda calculi, mainly extensions of the polymorphic lambda calculus. It is well-known that in polymorphic lambda calculus, many inductive data types can be defined (see e.g.[Böhm and Berarducci 1985] and [Girard et al. 1989]). In this paper we want to look at *how* functions on inductive types can be represented. Therefore, two ways of using the inductive building up of a type to define functions on that type are being distinguished, the *iterative* way and the *recursive* way. An *iterative* function is defined by induction on the building up of the type by defining the function value in terms of the previous values. A *recursive* function is also defined by induction, but now by defining the function value in terms of the previous values and the previous inputs. For functions on the natural numbers that is $h : \mathrm{Nat} \to A$, with $h(0) = c, h(n+1) = f(h(n))$ (for $c : A, f : A \to A$) is iterative and $h : \mathrm{Nat} \to A$, with

*Extended notes of a talk given at the BRA-LF meeting in Edinburgh, May 1991

†herman@cs.kun.nl

$h(0) = c, h(n+1) = g(h(n), n)$ (for $c : A, g : A \times \mathrm{Nat} \to A$) is recursive. If one has pairing, the recursive functions can be defined using just iteration, which was essentially already shown by [Kleene 1936]. But if we work in a typed lambda calculus where pairing is not surjective, this translation of recursion in terms of iteration becomes inefficient and sometimes impossible. Moreover, if the calculus also incorporates some predicate logic, one would like to use the inductivity in doing proofs, which is not always straightforward (or just impossible.) We shall not go into the latter topic here; there is still a lot of work to be done in relating the work presented here to systems like AF2 by Krivine and Parigot (connections may be found in [Parigot 1992]) and Coq ([Dowek e.a. 1991].)

This asks for an explicit scheme for recursion in typed lambda calculus, which yields for, say, the natural numbers the scheme of Gödels T. To see how this can be done in general for inductive types, we are going to define a categorical notion of recursion (just like 'initial algebra' categorically represents the notion of iteration). One of the trade-offs is that we can dualize all this to get a notion of *corecursion* on coinductive types. These categorical notions of recursion and corecursion have independently been found by Mendler (see [Mendler 1991]) who treats these constructions in Martin-Löf type theory with predicative universes. What we define as (co)recursive (co)algebras are what Mendler calls '(co)algebras that admit simple primitive recursion'. We shall always use the term 'recursion', because, although the function-definition-scheme has a strong flavour of primitive recursion, one can define many more functions in polymorphic lambda calculus then just the primitive recursive ones. Coinductive types were first described in [Hagino 1987a] and [Hagino 1987b], with only a scheme for coiteration and without corecursion. Here we give a quite straightforward extension of simply typed lambda calculus with recursive and corecursive types.

A very surprising result is that in a polymorphic framework, if we have a notion of recursive types which reflects our notion of recursive algebra, then we can define corecursive types that correspond to corecursive coalgebras. By duality, this also works the other way around. This result will be given here syntactically: We define a polymorphic lambda calculus with recursive and corecursive types (that straightforwardly represents the categorical notions of recursive algebra and corecursive coalgebra) and show that the scheme for recursive types can be defined from the scheme for corecursive types and vice versa. We also look at a system of recursive and corecursive types defined by [Mendler 1987] and show that with either the scheme for recursive types or the scheme for corecursive types, there is a recursive Φ-algebra and a corecursive Φ-coalgebra in the syntax for every syntactic functor Φ (where syntactic functors are positive type schemes.)

2 The categorical perspective

As said, we shall get our intuitions about inductive and coinductive types from the field of category theory. The main notions in category theory related to this issue come from [Lambek 1968].

Definition 2.1 *Let C be a category, T a functor from C to C.*

1. *A T-*algebra *in C is a pair (A, f), with A an object and $f : TA \to A$.*

2. *If (A, f) and (B, g) are T-algebra's,* a morphism from (A, f) to (B, g) *is a morphism $h : A \to B$ such that the following diagram commutes.*

$$\begin{tikzcd} TA \arrow[r, "f"] \arrow[d, "Th"'] \arrow[dr, phantom, "="] & A \arrow[d, "h"] \\ TB \arrow[r, "g"'] & B \end{tikzcd}$$

3. *A T-algebra (A, f) is* initial *if it is initial in the category of T-algebras, i.e. for every T-algebra (B, g) there's a unique h which makes the above diagram commute.*

In a category with products, coproducts and terminal object, the initial algebra of the functor $TX = 1 + X$ is the natural numbers object, for which we write $(\mathsf{Nat}, [\mathsf{Z}, \mathsf{S}])$. The initial algebra of $TX = 1 + (A \times X)$ is the object of finite lists over A, $(\mathsf{List}_A, [\mathsf{Nil}, \mathsf{Cons}])$. In this paper our pet-example of an initial algebra will be $(\mathsf{Nat}, [\mathsf{Z}, \mathsf{S}])$, which will be used to illustrate the properties we are interested in. First take a look at how the iterative and recursive functions can be defined on Nat. (The example immediately generalizes to arbitrary initial algebras.)

Example 2.2 1. *For $g : 1 + B \to B$ we write g_1 for $g \circ \mathsf{in}_1 : 1 \to B$ and g_2 for $g \circ \mathsf{in}_2 : B \to B$. The* iteratively defined morphism from g_1, g_2, $\mathsf{Elim} g_1 g_2$, *is defined as the unique morphism h which makes the diagram commute, i.e. $h \circ \mathsf{Z} = g_1$ and $h \circ \mathsf{S} = g_2 \circ h$.*

2. *For $g_1 : 1 \to B$, $g_2 : B \times \mathsf{Nat} \to B$, the* recursively defined morphism from g_1 *and* g_2 *is constructed as follows.*
There exists a unique h which makes the diagram

$$\begin{tikzcd} 1 + \mathsf{Nat} \arrow[r, "{[\mathsf{Z}, \mathsf{S}]}"] \arrow[d, "\mathsf{id} + h"'] \arrow[dr, phantom, "="] & \mathsf{Nat} \arrow[d, "h"] \\ 1 + (B \times \mathsf{Nat}) \arrow[r, "{\langle [g_1, g_2], [\mathsf{Z}, \mathsf{S} \circ \pi_2] \rangle}"'] & B \times \mathsf{Nat} \end{tikzcd}$$

commute. That is $h \circ [\mathsf{Z}, \mathsf{S}] = \langle [g_1, g_2], [\mathsf{Z}, \mathsf{S} \circ \pi_2] \rangle \circ \mathsf{id} + h$. If we write $h_1 = \pi_1 \circ h$ and $h_2 = \pi_2 \circ h$ we have the equalities

$$\begin{aligned} h_1 \circ \mathsf{Z} &= g_1, \\ h_1 \circ \mathsf{S} &= g_2 \circ h, \\ h_2 \circ \mathsf{Z} &= \mathsf{Z}, \\ h_2 \circ \mathsf{S} &= \mathsf{S} \circ h_2. \end{aligned}$$

Now $h_2 = \mathsf{id}_{Nat}$ by uniqueness and also $h = \langle h_1, h_2 \rangle$, so

$$\begin{aligned} h_1 \circ \mathsf{Z} &= g_1, \\ h_1 \circ \mathsf{S} &= g_2 \circ \langle h_1, \mathsf{id} \rangle. \end{aligned}$$

So h_1 satisfies the recursion equalities and we define

$$\mathsf{Rec} g_1 g_2 := h_1.$$

Definition 2.3 *Let C be a category, T a functor from C to C.*

1. *A T-*coalgebra in C *is a pair (A, f), with A an object and $f : A \to TA$.*

2. *If (A, f) and (B, g) are T-coalgebras,* a morphism from (B, g) to (A, f) *is a morphism $h : B \to A$ such that the following diagram commutes.*

$$\begin{tikzcd}
B \arrow[r, "g"] \arrow[d, "h"'] \arrow[dr, phantom, "="] & TB \arrow[d, "Th"] \\
A \arrow[r, "f"'] & TA
\end{tikzcd}$$

3. *A T-coalgebra (A, f) is* terminal *if it is terminal in the category of T-coalgebras, i.e. for every coalgebra (B, g) there's a unique h which makes the above diagram commute.*

Our pet example for terminal coalgebras is the one for $TX = \mathsf{Nat} \times X$, the object of infinite lists of natural numbers, for which we write $(\mathsf{Stream}, \langle \mathsf{H}, \mathsf{T} \rangle)$. We shall dualize the notions of iterative and recursive function to get *coiterative* and *corecursive* functions *to* Stream. (Again this example easily generalizes to the case for arbitrary terminal coalgebras.)

Example 2.4

1. *For $g : B \to \mathsf{Nat}{\times}B$, write g_1 for $\pi_1{\circ}g : B \to \mathsf{Nat}$ and g_2 for $\pi_2{\circ}g : B \to B$. The coiteratively defined morphism from g_1 and g_2, $\mathsf{Intro} g_1 g_2 : B \to \mathsf{Stream}$ is the (unique) morphism h for which the diagram commutes. That is, $H \circ h = g_1$ and $T \circ h = h \circ g_2$.*
   If j is a morphism from Nat to Nat, one can define the morphism from Stream to Stream which applies f to every point in the stream as $\mathsf{Intro}(j \circ \mathsf{H})T$. Note that it is not so straightforward to define (coiteratively) a morphism which replaces the head of a stream by, say, zero. This, however, can easily be done using corecursion.

2. *For $g_1 : B \to \mathsf{Nat}$, $g_2 : B \to B + \mathsf{Stream}$, the corecursively defined morphism from g_1 and g_2, $\mathsf{Corec} g_1 g_2$ is defined by $h \circ \mathsf{in}_1$, where h is the (unique) morphism which makes the diagram*

$$\begin{tikzcd}
B + \mathsf{Stream} \arrow[r, "{[\langle g_1, g_2 \rangle, \langle \mathsf{H}, \mathsf{in}_2 \circ \mathsf{T} \rangle]}"] \arrow[d, "h"'] \arrow[dr, phantom, "="] & \mathsf{Nat} \times (B + \mathsf{Stream}) \arrow[d, "\mathsf{id} \times h"] \\
\mathsf{Stream} \arrow[r, "{\langle \mathsf{H}, \mathsf{T} \rangle}"'] & \mathsf{Nat} \times \mathsf{Stream}
\end{tikzcd}$$

   commute. If we write $h_1 = h \circ \mathsf{in}_1, h_2 = h \circ \mathsf{in}_2$, then we have for h the following equations

$$\begin{aligned}
\mathsf{H} \circ h_2 &= \mathsf{H}, \\
\mathsf{T} \circ h_2 &= h_2 \circ \mathsf{T}, \\
\mathsf{H} \circ h_1 &= g_1, \\
\mathsf{T} \circ h_1 &= h \circ g_2.
\end{aligned}$$

Now $h_2 = \mathsf{id}$ *by uniqueness and also* $h = [h_1, h_2]$, *so*

$$\begin{aligned} \mathsf{H} \circ h_1 &= g_1, \\ \mathsf{T} \circ h_1 &= [h_1, \mathsf{id}] \circ g_2. \end{aligned}$$

These are the equations for corecursion; if $g_1 : B \to \mathsf{Nat}$ *and* $g_2 : B \to B + \mathsf{Stream}$*, then* $j : B \to \mathsf{Stream}$ *is* corecursively defined from g_1 and g_2 *if* $\mathsf{H} \circ j = g_1$ *and* $\mathsf{T} \circ j = [j, \mathsf{id}] \circ g_2$. *The function* $\mathsf{ZeroH} : \mathsf{Stream} \to \mathsf{Stream}$ *which changes the head of a stream into zero can now be defined as* $\mathsf{ZeroH} := \mathsf{Corec}(\mathsf{Z}\circ!)(\mathsf{in}_2 \circ \mathsf{T})$, *where* ! *is the unique morphism from* Stream *to* 1. *(Informally,* $\mathsf{Z}\circ!$ *is of course just* $\lambda s : \mathsf{Stream}.0$.*)*

As usual in categorical definitions, the definitions of initial algebra and terminal coalgebra split up in two parts, the 'existence part' (there's an h such that...) and the 'uniqueness part' (the h is unique.) In the following we shall sometimes refer to these two parts of the definition as the *existence property* and the *uniqueness property*.

In the typed lambda calculi that we shall consider, the inductive and coinductive types will not exactly represent initial algebras and terminal coalgebras. What the systems are lacking is the uniqueness property for the morphism h in 2.1, respectively 2.3. Algebras, respectively coalgebras, which only satisfy the existence property are called *weakly initial*, respectively *weakly terminal*.

Definition 2.5 *For* T *an endofunctor in a category* C, *The* T*-algebra, respectively* T*-coalgebra,* (A, f) *is* weakly initial, *respectively* weakly terminal, *if for every* T*-algebra, respectively* T*-coalgebra,* (B, g) *there exists an arrow* h *that makes the diagram in 2.1, respectively 2.3, commute.*

Remark 2.6 *The notion of weakly initial algebra is really weaker than that of initial algebra. For example in the category* **Set**, $(2\omega, [\mathsf{Z}, \mathsf{S}])$ *is a weakly initial* $(\lambda X.1 + X)$*-algebra, but also* $(2\omega, [\mathsf{Z}, \mathsf{S}'])$, *with* $\mathsf{S}'(n) = \mathsf{S}(n)$, $\mathsf{S}'(\omega + n) = n$ *is. (On weakly initial algebras, the behaviour of morphisms is only determined on the standard part of the algebra, that is in settheoretic terms, those elements that are constructed by finitely many times applying the constructor* f. *Initiality says that the algebra is standard.)*

As we made serious use of the uniqueness property in constructing the recursive and corecursive functions, it's interesting to see how much we can do in weak initial algebras and weak terminal coalgebras. The construction of the iterative and coiterative functions of examples 2.2 and 2.4 can be done in the same way; we only loose the uniqueness property of the iteratively defined function. The construction of recursive and corecursive functions in a weak framework is not so straightforward. We shall study again the examples of natural numbers and streams of natural numbers. Fix a category C, which has weak products and coproducts. (So we *do* have e.g. $\pi_1 \circ \langle t_1, t_2 \rangle = t_1$ and $[t_1, t_2] \circ \mathsf{in}_1 = t_1$, but not $\langle \pi_1 \circ t, \pi_2 \circ t \rangle = t$ and $[t \circ \mathsf{in}_1, t \circ \mathsf{in}_2] = t$.) It will turn out that weak products and coproducts will cause some extra restrictions on the definability of functions. Therefore we shall also study what happens if product and coproduct are *semi*, that is for products $\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle$ and for coproducts $h \circ [f, g] = [h \circ f, h \circ g]$. The reason for not considering the strong products and coproducts in these examples is that in the syntax of typed lambda calculi product and coproduct are usually weak or semi. (The notions of semi product and semi coproduct are taken from [Hayashi 1985].)

Example 2.7 *(Recursion on a weak natural numbers object) Let* Nat *be a weakly initial* $\lambda X.1+X$*-algebra. Consider the diagram in 2.2, where we defined recursion in terms of iteration and let* $h : \mathsf{Nat} \to B \times \mathsf{Nat}$ *be some morphism that makes the diagram commute, i.e.*

$$h \circ [\mathsf{Z}, \mathsf{S}] = \langle [g_1, g_2], [\mathsf{Z}, \mathsf{S} \circ \pi_2] \rangle \circ \mathrm{id} + h.$$

Applying projections to the left and injections to the right of the equation we obtain the following equalities (where $h_1 = \pi_1 \circ h$ *and* $h_2 = \pi_2 \circ h$*).*

$$\begin{array}{rcl} h_1 \circ \mathsf{Z} &=& g_1, \\ h_1 \circ \mathsf{S} &=& g_2 \circ h, \\ h_2 \circ \mathsf{Z} &=& \mathsf{Z}, \\ h_2 \circ \mathsf{S} &=& \mathsf{S} \circ h_2. \end{array}$$

Nat *doesn't satisfy the uniqueness properties, so not necessarily* $h_2 = \mathsf{id}_{Nat}$*but only*

$$h_2 \circ \mathsf{S}^n \circ \mathsf{Z} = \mathsf{S}^n \circ \mathsf{Z}$$

for every $n \in \mathbf{N}$*, where* S^n *denotes an* n*-fold composition of* S*. Now we would like to deduce*

$$\begin{array}{rcl} h_1 \circ \mathsf{Z} &=& g_1, \\ h_1 \circ \mathsf{S}^{n+1} \circ \mathsf{Z} &=& g_2 \circ \langle h_1, \mathsf{id} \rangle \circ \mathsf{S}^n \circ \mathsf{Z}, \end{array}$$

which says that h_1 *satisfies the recusion equations for the 'standard' natural numbers.*

- *For weak products this conclusion is only valid if* $g_2 = k \circ \pi_i$ *for some* $k : B \to B$ *or* $k : \mathsf{Nat} \to B$*. (Note that if* $g_2 = k \circ \pi_1$ *for some* $k : B \to B$*, then* h_1 *is just iteratively defined from* g_1 *and* k*, so only the case for* $g_2 = k \circ \pi_2$ *gives us really new functions, for instance the predecessor.)*
- *For semi products this conclusion is only valid for* $g_2 = k \circ \langle \pi_1, \pi_2 \rangle$ *for some* $k : B \times \mathsf{Nat} \to B$*, which is not a serious restriction: Just replace* g_2 *by* $g_2 \circ \langle \pi_1, \pi_2 \rangle$*.*

Example 2.8 *(Corecursion on a weak stream object) Let* Stream *be a weakly terminal* $\lambda X.\mathsf{Nat} \times X$*-coalgebra. Consider the diagram in 2.4, where we defined corecursion in terms of coiteration and let* $h : (B + \mathsf{Stream}) \to \mathsf{Stream}$ *be some morphism that makes the diagram commute. Write* $h_1 = h \circ \mathsf{in}_1$ *and* $h_2 = h \circ \mathsf{in}_2$*. We have the following equalities.*

$$\begin{array}{rcl} \mathsf{H} \circ h_2 &=& \mathsf{H}, \\ \mathsf{T} \circ h_2 &=& h_2 \circ \mathsf{T}, \\ \mathsf{H} \circ h_1 &=& g_1, \\ \mathsf{T} \circ h_1 &=& h \circ g_2. \end{array}$$

Now we can not conclude $h \circ \mathsf{in}_2 = \mathsf{id}$*, because we don't have uniqueness, but we do have*

$$\mathsf{H} \circ \mathsf{T}^n \circ h_2 = \mathsf{H} \circ \mathsf{T}^n,$$

that is h_2 *is the identity on the 'standard' part of the stream (those points that can be obtained by finitely many applications of* H *or* T*.) Again we would like to conclude*

$$\begin{array}{rcl} \mathsf{H} \circ h_1 &=& g_1, \\ \mathsf{H} \circ \mathsf{T}^{n+1} \circ h_1 &=& \mathsf{H} \circ \mathsf{T}^n \circ [h_1, \mathrm{id}] \circ g_2, \end{array}$$

that is h_1 *satisfies the corecursion equations for the 'standard' part of the stream.*

- *For weak coproducts this conclusion is only valid if* $g_2 = \mathsf{in}_i \circ k$ *for some* $k : B \to B$ *or* $k : B \to \mathsf{Stream}$. *(Note that if* $g_2 = \mathsf{in}_1 \circ k$ *for some* $k : B \to B$*, then* h_1 *is just coiteratively defined from* g_1 *and* k*, so only the case for* $g_2 = \mathsf{in}_2 \circ k$ *gives us really new functions, like for instance the function* ZeroH.*)*
- *For semi coproducts this conclusion is only valid if* $g_2 = [\mathsf{in}_1, \mathsf{in}_2] \circ k$ *for some* $k : B \to B + \mathsf{Stream}$. *again this is not a serious restriction: Just replace* g_2 *by* $[\mathsf{in}_1, \mathsf{in}_2] \circ g_2$.

For the morphism $\mathsf{ZeroH} : \mathsf{Stream} \to \mathsf{Stream}$ *which replaces the head by zero, defined in 2.4 by* $\mathsf{Corec}(\mathsf{Z}\circ !)(\mathsf{in}_2 \circ \mathsf{T})$*, we now have (for either weak or semi coproducts)*

$$\begin{aligned} \mathsf{H} \circ \mathsf{ZeroH} &= \mathsf{Z}, \\ \mathsf{H} \circ \mathsf{T}^{n+1} \circ \mathsf{ZeroH} &= \mathsf{H} \circ \mathsf{T}^{n+1}, \end{aligned}$$

so ZeroH *works fine on the standard part of the stream. That one can not, in general, define a morphism* ZeroH *such that* $\mathsf{T} \circ \mathsf{ZeroH} = \mathsf{T}$ *will be shown later, when we look at these examples in polymorphic lambda calculus which is an instance of a category with weakly initial algebras and weakly terminal coalgebras, semi products and weak coproducts.*

Remark 2.9 *With strong products and coproducts we would have similar problems in defining recursion and corecursion. The recursion equations would only be valid for the standard natural numbers and the corecursion equations would only be valid for the standard part of streams. The only advantage would be that the* $g_2 : B \times \mathsf{Nat} \to B$*, respectively the* $g_2 : B \to B + \mathsf{Stream}$ *can be taken arbitrarily.*

In Section 4 the polymorphic lambda calculus will be considered in which inductive and coinductive types can be defined which correspond to weakly initial algebras and weakly terminal coalgebras. It will be shown that recursion in that calculus is problematic from a point of view of efficiency. One solution could be to strengthen the reduction rules to get a stronger (extensional) equality. However, it's not possible to add some relatively easy reduction rules to the syntax to obtain the uniqueness property of initiality and terminality. (We can't say in an easy way that the only objects of a structure are the standard ones.) This is because the equality of (primitive) recursive functions can not be decided by an easy (decidable) equality. We can do something different, namely say that our functions should behave on the non-standard part as they behave on the standard part. Categorically, this can be obtained by strengthening the notion of weakly initial algebra and weakly terminal coalgebra a little bit, such that recursion 'works'. (That is for $\mathbf{N}$, for $c : A, g : A \times \mathrm{Nat} \to A$ there is a function $h : \mathrm{Nat} \to A$, with $h(0) = c$ and $h(n+1) = g(h(n), n)$.) These new notions will be called *recursive algebra* and *corecursive coalgebra*. The definitions are not difficult if one understands what makes it possible to define (co)recursion, in terms of (co)iteration.

Let in the following C be a category with weak products and weak coproducts and T a functor from C to C.

Definition 2.10 *(A, f) is a* recursive T-algebra *if (A, f) is a T-algebra and for every $g : T(X \times A) \to X$ there exists an $h : A \to X$ such that the following diagram commutes.*

$$\begin{tikzcd} TA \arrow[r, "f"] \arrow[d, "T(\langle h, \mathsf{id}\rangle)"'] \arrow[dr, phantom, "="] & A \arrow[d, "h"] \\ T(X \times A) \arrow[r, "g"'] & X \end{tikzcd}$$

Notice that this is the same as saying that (A, f) is weakly initial and that moreover, in the diagram for defining recursion in terms of iteration, $h_2 = \mathsf{id}$. (See 2.2)

Definition 2.11 *(A, f) is a* corecursive T-coalgebra *if (A, f) is a T-coalgebra and for every $g : X \to T(X + A)$ there exists an $h : X \to A$ such that the following diagram commutes.*

$$\begin{tikzcd} X \arrow[r, "g"] \arrow[d, "h"'] \arrow[dr, phantom, "="] & T(X + A) \arrow[d, "T([h, \mathsf{id}])"] \\ A \arrow[r, "f"'] & TA \end{tikzcd}$$

Again this is the same as saying that (A, f) is a weakly terminal T-coalgebra and that moreover, in the diagram for defining corecursion in terms of coiteration, $h_2 = \mathsf{id}$. (See 2.4)

When talking about weakly initial or recursive T-algebras and weakly terminal or corecursive T-coalgebras, it is convenient to denote the h that makes the diagram commute as a function of g. So we shall denote a weakly initial T-algebra by (A, f, Elim), where $\mathsf{Elim}g$ denotes a morphism h in 2.5 that makes the diagram commute. Similarly, we write (A, f, Intro) for a weakly terminal T-coalgebra, (A, f, Rec) for a recursive T-algebra and (A, f, Corec) for a corecursive T-coalgebra.

Examples 2.12

1. *If $(\mathsf{Nat}, [\mathsf{Z}, \mathsf{S}], \mathsf{Rec})$ is a recursive $\lambda X.1 + X$-algebra, Rec is a recursor on Nat: For $[g_1, g_2] : 1 + (X \times \mathsf{Nat}) \to X$,*

$$\begin{aligned} \mathsf{Rec}[g_1, g_2] \circ \mathsf{Z} &= g_1, \\ \mathsf{Rec}[g_1, g_2] \circ \mathsf{S} &= g_2 \circ \langle \mathsf{Rec}[g_1, g_2], \mathsf{id}\rangle, \end{aligned}$$

so $\mathsf{Rec}[g_1, g_2]$ is the recursively defined function from g_1 and g_2. We can define $\mathsf{P} := \mathsf{Rec}[\mathsf{Z}, \pi_2]$ and we have

$$\begin{aligned} \mathsf{P} \circ \mathsf{Z} &= \mathsf{Z}, \\ \mathsf{P} \circ \mathsf{S} &= \mathsf{id}. \end{aligned}$$

2. *If $(\mathsf{Stream}, \langle \mathsf{H}, \mathsf{T}\rangle, \mathsf{Corec})$ is a corecursive $\lambda X.\mathsf{Nat} \times X$-coalgebra. Then for $\langle g_1, g_2\rangle : X \to \mathsf{Nat} \times (X + \mathsf{Stream})$, the function $\mathsf{Corec}\langle g_1, g_2\rangle$ satisfies*

$$\begin{aligned} H \circ \mathsf{Corec}\langle g_1, g_2\rangle &= g_1, \\ T \circ \mathsf{Corec}\langle g_1, g_2\rangle &= [\mathsf{Corec}\langle g_1, g_2\rangle, \mathsf{id}] \circ g_2, \end{aligned}$$

so $\mathsf{Corec}\langle g_1, g_2\rangle$ *is the corecursively defined function from* g_1 *and* g_2. *We can define*

$$\mathsf{ZeroH} := \mathsf{Corec}\langle \mathsf{Z}\circ !, \mathsf{in}_2 \circ \mathsf{T}\rangle$$

with

$$\begin{aligned}\mathsf{H} \circ \mathsf{ZeroH} &= \mathsf{Z}\circ !,\\ \mathsf{T} \circ \mathsf{ZeroH} &= \mathsf{T}.\end{aligned}$$

3 Extending simply typed lambda calculus with inductive and coinductive types

In his thesis ([Hagino 1987a]) Hagino derives from the notions of initial algebra and terminal coalgebra an extension of simply typed lambda calculus, which he calls categorical data types. This amounts to adding two schemes for defining a new type from a covariant functor from types to types. (In the notation of these schemes below we follow [Wraith 1989].) These new types come together with some constants and reduction rules. A covariant functor from types to types in $\lambda\rightarrow$ is a *positive type scheme* $\Phi(\alpha)$, that is a type Φ in which the free variable α occurs *positively*. (The type variable α occurs *positively* in the type Φ if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi \equiv \alpha$ or if $\Phi \equiv \Phi_1{\to}\Phi_2$ and α occurs negatively in Φ_1, positively in Φ_2. The type variable α occurs *negatively* in Φ if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi \equiv \alpha$ or if $\Phi \equiv \Phi_1{\to}\Phi_2$ and α occurs negatively in Φ_2, positively in Φ_1.) If $\Phi(\alpha)$ is a type scheme, with $\Phi(\tau)$ we mean the type Φ with τ substituted for α. If there's no ambiguity to which type variable α we're referring, we just write Φ in stead of $\Phi(\alpha)$.

A positive, respectively negative type scheme Φ can be applied to a function $f{:}\tau{\to}\rho$, obtaining $\Phi(f){:}\Phi(\rho){\to}\Phi(\tau)$, respectively $\Phi(f){:}\Phi(\tau){\to}\Phi(\rho)$ by *lifting*: $\alpha(f) \equiv f$, if $\alpha \notin \mathrm{FV}(\Phi)$, $\Phi(f) \equiv id_\Phi$ and if α occurs negatively in Φ_1, positively in Φ_2 then

$$\begin{aligned}(\Phi_1{\to}\Phi_2)(f) &\equiv \lambda x{:}\Phi_1(\rho){\to}\Phi_2(\rho).\lambda y{:}\Phi_1(\tau).\Phi_2(f)(x(\Phi_1(f)y)),\\ (\Phi_2{\to}\Phi_1)(f) &\equiv \lambda x{:}\Phi_2(\tau){\to}\Phi_1(\tau).\lambda y{:}\Phi_2(\rho).\Phi_1(f)(x(\Phi_2(f)y)).\end{aligned}$$

Definition 3.1 *Let* $\Phi_1, \Phi_2, \ldots, \Phi_n$ *be types in the simply typed lambda calculus in which the typevariable* α *occurs positively. The* sum scheme *for constructing data types is the following.*

$\sigma =$ **sum** α **with constructors**

$$\begin{array}{lll} c_1 & : & \Phi_1{\to}\alpha\\ c_2 & : & \Phi_2{\to}\alpha\\ & \vdots & \\ c_n & : & \Phi_n{\to}\alpha \end{array}$$

end

A declaration of a type σ *using this sum scheme gives rise to an extension of the language of* $\lambda\rightarrow$ *with*

1. *a closed type* σ
2. *constants* $c_i{:}\Phi_i(\sigma){\to}\sigma$ *for* $1 \le i \le n$,
3. *for every type* τ, $\mathit{Elim}_\tau{:}(\Phi_1(\tau){\to}\tau){\to}(\Phi_2(\tau){\to}\tau){\to}\ldots{\to}(\Phi_n(\tau){\to}\tau){\to}\sigma{\to}\tau$.

The reduction relation is extended with the rule

$$\mathit{Elim}_\tau M_1 M_2 \ldots M_n(c_i t) \longrightarrow M_i(\Phi_i(\mathit{Elim}_\tau M_1 \ldots M_n)t)$$

An easy example of a type defined by the sum scheme is $\sigma{+}\tau$ (for σ and τ types), representing the the disjoint sum of σ and τ.

$\sigma + \tau =$ **sum** α **with constructors**

  inl : $\sigma{\to}\alpha$

  inr : $\tau{\to}\alpha$

end

with inl : $\sigma{\to}\sigma + \tau$, inr : $\tau{\to}\sigma + \tau$ and for $M_1{:}\sigma{\to}\rho$, $M_2{:}\tau{\to}\rho$, $[M_1, M_2] : \sigma + \tau{\to}\rho$.

The sequence of type schemes in the sum scheme can also be empty, allowing us to define the unit type by

$1 =$ **sum** α **with constructors**

  $*$: α

end

We have $* : 1$ and for any $t : \tau$, $!(t) : 1{\to}\tau$ with $!(t)(*) \longrightarrow t$.

Definition 3.2 *Let $\Phi_1, \Phi_2, \ldots, \Phi_n$ be types in the simply typed lambda calculus in which the typevariable α occurs positively. The* product scheme *for constructing new data types is the following.*

$\sigma =$ **product** α **with destructors**

  d_1 : $\alpha{\to}\Phi_1$

  d_2 : $\alpha{\to}\Phi_2$

  $\vdots$

  d_n : $\alpha{\to}\Phi_n$

end

A declaration of a type σ using this product scheme gives rise to an extension of the language of $\lambda\rightarrow$ with

1. *a closed type σ,*
2. *constants $d_i{:}\sigma{\to}\Phi_i(\sigma)$, for $1 \leq i \leq n$,*
3. *for every type τ, $\mathit{Intro}_\tau{:}(\tau{\to}\Phi_1(\tau)){\to}(\tau{\to}\Phi_2(\tau)){\to}\ldots{\to}(\tau{\to}\Phi_n(\tau)){\to}\tau{\to}\sigma$.*

The reduction relation is extended with the rule

$$d_i(\mathit{Intro}_\tau M_1 M_2 \ldots M_n t) \longrightarrow \Phi_i(\mathit{Intro}_\tau M_1 \ldots M_n)(M_i t)$$

The straightforward example of a type defined by the product scheme is $\sigma \times \tau$ (for σ and τ types), representing the the product of σ and τ.

$\sigma \times \tau =$ **product** α **with destructors**

  fst : $\alpha{\to}\sigma$

  snd : $\alpha{\to}\tau$

end

with fst : $\sigma \times \tau{\to}\sigma$, snd : $\sigma \times \tau{\to}\tau$ and for $M_1{:}\rho{\to}\sigma$, $M_2{:}\rho{\to}\tau$, $<M_1, M_2> : \rho{\to}\sigma \times \tau$.

Remark 3.3 *The type σ defined by the sum scheme from $\Phi_1(\alpha), \ldots \Phi_n(\alpha)$, will be denoted by $\mu\alpha.(\Phi_1(\alpha) + \ldots + \Phi_n(\alpha))$. The type σ defined by the product scheme from $\Phi_1(\alpha), \ldots \Phi_n(\alpha)$, will be denoted by $\nu\alpha.(\Phi_1(\alpha) \times \ldots \times \Phi_n(\alpha))$. This is also how these types should be read: as (weakly) initial algebras of $TX = \Phi_1(X) + \ldots + \Phi_n(X)$ and (weakly) terminal coalgebras of $TX = \Phi_1(X) \times \ldots \times \Phi_n(X)$, respectively. (So dualising is of course not the same as reversing all the arrows in a sum scheme to obtain a product scheme!)*

Definition 3.4 *$\lambda \to^{ind}$ is the simply typed lambda calculus extended with sum scheme and product scheme.*

Example 3.5 *The iterative functions on an inductive type can be straightforwardly defined by the Elim construct. Write Nat for $\mu\alpha.1 + \alpha$, then for $c{:}\tau$ and $f{:}\tau\to\tau$, Elim$(\lambda z.c)f{:}\mathrm{Nat}\to\tau$ is the iteratively defined function from c and f. The recursive functions can be defined by translating recursion in terms of iteration as is done in 2.2. For $c{:}\tau$, $g{:}\tau\to\mathrm{Nat}\to\tau$, define Reccg $:= \mathrm{fst} \circ \mathrm{Elim}(\lambda z.\langle c, 0\rangle)(\langle \lambda x.g(\mathrm{fst}x)(\mathrm{snd}x), S \circ \mathrm{snd}\rangle)$ and we have*

$$\begin{aligned}\mathrm{Rec}cg0 &\twoheadrightarrow c,\\ \mathrm{Rec}cg(S^{n+1}(0)) &\twoheadrightarrow g(\mathrm{Rec}cg(S^n 0))(\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle c,0\rangle)(\langle \lambda x.g(\mathrm{fst}x)(\mathrm{snd}x), S\circ\mathrm{snd}\rangle)(S^n(0)))).\end{aligned}$$

This recursor only works for terms of type Nat which are of the form $S^n 0$, but moreover, it is quite inefficient (compared to, for instance, the recursor in Gödels T).

Proposition 3.6 *The predecessor function of type* Nat$\to$Nat*, defined in terms of iteration in $\lambda \to^{ind}$ computes the predecessor of a numeral $n + 1$ in $3n + 2$ steps.*

Proof The predecessor function P is the $\beta\eta$-normal form of Rec0$(\lambda xy.y)$, so

$$P \equiv \mathrm{fst} \circ \mathrm{Elim}_{\mathrm{Nat}\times\mathrm{Nat}}(\lambda z.\langle 0,0\rangle)(\langle \mathrm{snd}, S\circ\mathrm{snd}\rangle).$$

Now

$$P(S^{n+1}0) \twoheadrightarrow_2 \mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle \mathrm{snd}, S\circ\mathrm{snd}\rangle)(S^n(0))),$$
$$\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle \mathrm{snd}, S\circ\mathrm{snd}\rangle)(0)) \twoheadrightarrow_3 0$$

and with induction one proves that

$$\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle \mathrm{snd}, S\circ\mathrm{snd}\rangle)(S^{n+1}(0))) \twoheadrightarrow_3 S(\mathrm{snd}(\mathrm{Elim}(\lambda z.\langle 0,0\rangle)(\langle \mathrm{snd}, S\circ\mathrm{snd}\rangle)(S^n(0)))).$$

Proposition 3.7 *In $\lambda \to^{ind}$ there is no term $P{:}\mathrm{Nat}\to\mathrm{Nat}$ with*

$$P(S0) = 0 \textit{ and}$$

$$P(Sx) = x$$

for x a variable of type Nat.

Proof This follows by the Church-Rosser property for reduction in $\lambda \to^{ind}$. (See [Hagino 1987b].) If $P(Sx) = x$, then $P(Sx) \twoheadrightarrow x$. Analyzing the possible structure of P one can conclude that if $P(Sx) \twoheadrightarrow x$, then not at the same time $P(S0) \twoheadrightarrow 0$. This proposition is also an immediate corollary of the same proposition for system F in 4.

If one tries to do corecursion on the coinductive types in $\lambda \to^{ind}$, a similar situation occurs. For Stream $:= \nu\alpha.\mathrm{Nat}\times\alpha$, one can define ZeroH:Stream$\to$Stream which replaces the head by 0 using the definable corecursion in weakly terminal coalgebras. We do not have $T(\mathrm{ZeroH}s) = Ts$, for s a Stream, but just $H(\mathrm{ZeroH}s) = 0$ and $H(T^{n+1}(\mathrm{ZeroH}s)) = H(T^{n+1}s)$. One can also show that there can be no term ZeroH:Stream$\to$Stream such that $T(\mathrm{ZeroH}s) = s$ for a variable s:Stream. (Using the Church-Rosser property or as a corollary of the same proposition for system F.)

There are of course ways to strengthen the equalities of the sum and product scheme to get real recursion and corecursion. The initiality can be restored totally by adding the conditional rewrite rule

If $h(c_it) = M_i(\sigma_i[h]t)$ for $1 \leq i \leq n$ and M_i and t of appropriate type, then $h \longrightarrow \mathrm{Elim}_\tau M_1 \dots M_n$.

However, conditional rewrite rules are metatheoretically very complicated (the rewriting depends on the typing and on the previously generated equality.) Another alternative, which restores part of the unicity is to add a rewrite rule

$$Elim_\sigma c_1 \dots c_n \longrightarrow Id_\sigma,$$

for
$\sigma =$ **sum** α **with constructors**

$$\begin{array}{lcl} c_1 & : & \Phi_1 \to \alpha \\ c_2 & : & \Phi_2 \to \alpha \\ & \vdots & \\ c_n & : & \Phi_n \to \alpha \end{array}$$

end

This is not enough to obtain a recursive algebra, because the Elim constructor doesn't automatically commute with pairing. One has to add

$$\mathrm{snd} \circ \mathrm{Elim}_{\tau\times\sigma}\langle g_1, c_1 \circ \Phi_1(\mathrm{snd})\rangle, \dots, \langle g_n, c_n \circ \Phi_n(\mathrm{snd})\rangle \longrightarrow \mathrm{Elim}_\sigma c_1 \dots c_n.$$

In the proof of Proposition 3.6, we then have that $\mathrm{Rec}0(\lambda xy.y)(S^{n+1}0) \twoheadrightarrow S^N(0)$ in a constant number of steps. In this case it is of course better to take $\times$ (and $+$ if we add similar rules for the product scheme) as primitive type constructors. The new reduction rule is not a very pretty one.

We can also follow the categorical definitions of recursion and corecursion and strengthen the sum and product schemes themselves. (Again it is best to take $\times$ and $+$ as primitives.) For the sum scheme this would lead to the type σ with the same constructors and further

1. for every type τ, $\mathrm{Rec}_\tau{:}(\sigma_1(\tau\times\sigma)\to\tau)\to(\sigma_2(\tau\times\sigma)\to\tau)\to\dots(\sigma_n(\tau\times\sigma)\to\tau)\to\sigma\to\tau$,
2. the reduction rule $\mathrm{Rec}_\tau M_1 M_2 \dots M_n(c_it) \longrightarrow M_i(\sigma_i[\langle \mathrm{Rec}_\tau M_1 \dots M_n, \mathrm{id}\rangle]t)$

For the product scheme we would also get the same type σ with the same destructors and further

1. for every type τ, $\mathrm{Corec}_\tau{:}(\tau\to\sigma_1(\tau+\sigma))\to(\tau\to\sigma_2(\tau+\sigma))\to\dots(\tau\to\sigma_n(\tau+\sigma))\to\tau\to\sigma$,
2. the reduction rule $d_i(\mathrm{Corec}_\tau M_1 M_2 \dots M_n t) \longrightarrow \sigma_i[[\mathrm{id}, \mathrm{Corec}_\tau M_1 \dots M_n]](M_i t)$.

Call the system $\lambda\to^{ind}$ with modified sum and product scheme as above $\lambda\to^{rec}$. Without proof we give the following proposition.

Proposition 3.8 *In the system $\lambda\to^{rec}$ the inductive types are recursive algebras and the coinductive types are corecursive coalgebras. (For the appropriate functors.)*

4 The polymorphic lambda calculus

We just give the rules to fix our notation and shall not go into the system further, assuming it is familiar. We write $\times$ and $+$ for the definable weak product and coproduct: $\sigma \times \tau \equiv \forall\alpha.(\sigma{\to}\tau{\to}\alpha){\to}\alpha$ and $\sigma + \tau \equiv \forall\alpha.(\sigma{\to}\alpha){\to}(\tau{\to}\alpha){\to}\alpha$.) We could also have added $\times$ and $+$ as new type constructors with extra rules turning them into a weak product and coproduct. This however is inconvenient: The added $\times$ and $+$ would not be functorial (e.g. $\times$: Types $\times$ Types$\to$Types does not preserve identities and composition), whereas the definable $\times$ and $+$ are functorial by construction if we assume an η-reduction rule. (See Definition 4.2 and the discussion.)

Definition 4.1 *1. The set of types of F, $\mathbf{T}$, is defined by the following abstract syntax.*

$$\mathbf{T} ::= \mathit{TypVar} \mid \mathbf{T}{\to}\mathbf{T} \mid \forall \mathit{TypVar}.\mathbf{T}$$

2. The expressions of F, T, are defined by the following abstract syntax.

$$T ::= \mathit{Var} \mid TT \mid T\mathbf{T} \mid \lambda \mathit{Var}{:}\mathbf{T}.T \mid \Lambda \mathit{TypVar}.T$$

3. A context *is a sequence of* declarations *$x{:}\sigma$ ($x \in \mathit{Var}$ and $\sigma \in \mathbf{T}$), where it is assumed that if $x{:}\sigma$ and $y{:}\tau$ are different declarations in the same context, then $x \not\equiv y$.*

4. The typing rules for deriving judgements of the form $\Gamma \vdash M{:}\sigma$ for Γ a context, M an expression and σ a type, are the following.

- *If $x{:}\sigma$ is in Γ, then $\Gamma \vdash x{:}\sigma$,*
- $\dfrac{\Gamma \vdash M{:}\sigma{\to}\tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash MN{:}\tau}$ $\qquad$ $\dfrac{\Gamma, x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x{:}\sigma.M{:}\sigma{\to}\tau}$
- $\dfrac{\Gamma \vdash M{:}\forall\alpha.\sigma}{\Gamma \vdash M\tau{:}\sigma[\tau/\alpha]}$ *if $\tau \in \mathbf{T}$.* $\qquad$ $\dfrac{\Gamma \vdash M{:}\sigma}{\Gamma \vdash \Lambda\alpha.M{:}\forall\alpha.\sigma}$ *if $\alpha \notin FTV(\Gamma)$.*

FTV denotes the set of free type variables (TypVar.)

5. The one step reduction rules are the following.

- $(\lambda x{:}\sigma.M)N \longrightarrow_\beta M[N/x]$,
- $\lambda x{:}\sigma.Mx \longrightarrow_\eta M$ *if $x \notin FV(M)$,*
- $(\Lambda\alpha.M)\tau \longrightarrow_\beta M[\tau/\alpha]$,
- $\Lambda\alpha.M\alpha \longrightarrow_\eta M$ *if $\alpha \notin FTV(M)$.*

FV denotes the free term variables (Var.) One step reduction, $\longrightarrow$, is defined as the union of $\longrightarrow_\beta$ and $\longrightarrow_\eta$. The relations $\twoheadrightarrow$ and $=$ are respectively defined as the transitive, reflexive and the transitive, reflexive, symmetric closure of $\longrightarrow$.

Here, $t'[t/u]$ denotes the substitution of t for the variable u in t'. Substitution is done with the usual care, renaming bound variables such that no free variable becomes bound after substitution.

Type variables will be denoted by the lower case Greek characters α, β and γ, term variables will be denoted by lower case Roman characters. The set of expressions typable in the context Γ with type σ is denoted by $\mathrm{Term}(\sigma, \Gamma)$.

We want to discuss categorical notions like weak initiality in the syntax and therefore define need a syntactic notion of functor. This will be covered by the (well-known) notion of positive or negative type scheme.

Definition 4.2 *1. A* type scheme in F *is a type* $\Phi(\alpha)$ *where* α *marks all occurrences (possibly none) of* α*.*

2. *A type scheme* $\Phi(\alpha)$ *can be* positive *or* negative *(but also none of the both), which is defined by induction on the structure of* $\Phi(\alpha)$ *as follows.*
   - *(a) If* $\alpha \notin FTV(\Phi(\alpha))$*, then* $\Phi(\alpha)$ *is* positive *and* negative*,*
   - *(b) if* $\Phi(\alpha) \equiv \alpha$ *then* $\Phi(\alpha)$ *is* positive*,*
   - *(c) if* $\Phi(\alpha) \equiv \Phi_1(\alpha) \to \Phi_2(\alpha)$*, then* $\Phi(\alpha)$ *is* positive *if* $\Phi_1(\alpha)$ *is negative and* $\Phi_2(\alpha)$ *is positive,* $\Phi(\alpha)$ *is* negative *if* $\Phi_1(\alpha)$ *is positive and* $\Phi_2(\alpha)$ *is negative,*
   - *(d) if* $\Phi(\alpha) \equiv \forall\beta.\Phi'(\alpha)$ *then* $\Phi(\alpha) \equiv \forall\beta.\Phi'(\alpha)$ *is* positive *(resp.* negative*) if* $\Phi'(\alpha)$ *is positive (resp. negative) .*

3. *A positive (resp. negative) type scheme* $\Phi(\alpha)$ *works covariantly (resp. contravariantly) on a term* $f{:}\sigma \to \tau$*, obtaining a term* $\Phi(f)$ *of type* $\Phi(\sigma) \to \Phi(\tau)$ *(resp.* $\Phi(\tau) \to \Phi(\sigma)$*), by* lifting*, defined inductively as follows. (Let* $f{:}\sigma \to \tau$*.)*
   - *(a) If* $\alpha \notin FTV(\Phi(\alpha))$*, then* $\Phi(f) := \mathrm{id}_{\Phi(\alpha)}$*,*
   - *(b) if* $\Phi(\alpha) \equiv \alpha$ *then* $\Phi(f) := f$*,*
   - *(c) if* $\Phi(\alpha) \equiv \Phi_1(\alpha) \to \Phi_2(\alpha)$*, then, if* $\Phi(\alpha)$ *is positive,* $\Phi(f) := \lambda x{:}\Phi_1(\sigma) \to \Phi_2(\sigma).\lambda y{:}\Phi_1(\tau).\Phi_2(f)(x(\Phi_1(f)y)))$*, if* $\Phi(\alpha)$ *is negative,* $\Phi(f) := \lambda x{:}\Phi_2(\tau) \to \Phi_1(\tau).\lambda y{:}\Phi_2(\sigma).\Phi_1(f)(x(\Phi_2(f)y)))$*,*
   - *(d) if* $\Phi(\alpha) \equiv \forall\beta.\Phi'(\alpha)$*, then, if* $\Phi(\alpha)$ *is positive,* $\Phi(f) := \lambda x{:}\Phi(\sigma).\lambda\beta.\Phi'(f)(x\beta)$*, if* $\Phi(\alpha)$ *is negative, then* $\Phi(f) := \lambda x{:}\Phi(\tau).\lambda\beta.\Phi'(f)(x\beta)$*.*

It is easy to check that the lifting preserves identity and composition: $\Phi(\mathrm{id}) = \mathrm{id}$ and if $\Phi(\alpha)$ is positive then $\Phi(f \circ g) = \Phi(f) \circ \Phi(g)$, if $\Phi(\alpha)$ is negative then $\Phi(f \circ g) = \Phi(g) \circ \Phi(f)$. This also works for type schemes containing $\times$ or $+$, if we interpret $\times$ and $+$ as the definable weak product and coproduct:

$$
\begin{aligned}
\sigma \times \tau &=: \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha, \\
\mathrm{fst} &:= \lambda x{:}\sigma \times \tau.x\sigma(\lambda y{:}\sigma.\lambda z{:}\tau.y), \\
\mathrm{snd} &:= \lambda x{:}\sigma \times \tau.x\tau(\lambda y{:}\sigma.\lambda z{:}\tau.z), \\
<f, g> &:= \lambda z{:}\rho.\lambda\alpha.\lambda k{:}\sigma \to \tau \to \rho.k(fz)(gz), \\
&\quad\ \text{for } f{:}\sigma \to \rho, g{:}\tau \to \rho, \\
\sigma + \tau &=: \forall\alpha.(\sigma \to \alpha) \to (\tau \to \alpha) \to \alpha, \\
\mathrm{inl} &:= \lambda x{:}\sigma.\lambda\alpha.\lambda f : \sigma \to \alpha.\lambda g{:}\tau \to \alpha.fx, \\
\mathrm{inr} &:= \lambda x{:}\sigma.\lambda\alpha.\lambda f : \sigma \to \alpha.\lambda g{:}\tau \to \alpha.gx, \\
[f, g] &:= \lambda z{:}\sigma + \tau.z\rho f g, \\
&\quad\ \text{for } f{:}\rho \to \sigma, g{:}\rho \to \tau.
\end{aligned}
$$

It should be remarked here that if one lifts $f{:}\sigma\to\tau$ via a type scheme $\Phi(\alpha) \equiv \Phi_1(\alpha) \times \Phi_2(\alpha)$ (respectively $\Phi(\alpha) \equiv \Phi_1(\alpha) + \Phi_2(\alpha)$) according to Definition 4.2, this does not give the (expected) result $\Phi(f) = \lambda x{:}\Phi(\sigma).<\Phi_1(f)(\mathrm{fst}x), \Phi_2(f)(\mathrm{snd}x)>$ (respectively $\Phi(f) = [\mathrm{inl} \circ \Phi_1(f), \mathrm{inr} \circ \Phi_2(f)]$.) If we take the latter definition for lifting a function via a product or sum, this doesn't yield functoriality of $\times$ and $+$. We introduce some new notation to denote this lifting via $\times$ and $+$.

Definition 4.3 *Let $f{:}\sigma\to\tau$, $g{:}\mu\to\rho$.*

1. $\mathsf{times}fg : \sigma \times \mu \to \tau \times \rho$ *is defined by*

$$\mathsf{times}fg := \lambda z{:}\sigma \times \mu.\lambda\beta.\lambda y{:}\tau\to\rho\to\beta.z\beta(\lambda p{:}\sigma.\lambda q{:}\mu.y(fp)(gq)).$$

2. $\mathsf{plus}fg : \sigma + \mu \to \tau + \rho$ *is defined by*

$$\mathsf{plus}fg := \lambda z{:}\sigma + \mu.\lambda\beta.\lambda y_1{:}\tau\to\beta.y_2{:}\rho\to\beta.z\beta(y_1 \circ f)(y_2 \circ g).$$

Now for $f{:}\sigma\to\tau$, if $\Phi(\alpha) = \Phi_1(\alpha) \times \Phi_2(\alpha)$ then $\Phi(f) = \mathsf{times}(\Phi_1(f))(\Phi_2(f))$ and if $\Psi(\alpha) = \Psi_1(\alpha) + \Psi_2(\alpha)$ then $\Psi(f) = \mathsf{plus}(\Psi_1(f))(\Psi_2(f))$. Let's state some more easy facts about times and plus, some of which will be used later.

Fact 4.4 *For f, g, h and k of the right type we have.*

1. $\mathsf{plus}fg \circ \mathrm{inl} = \mathrm{inl} \circ f$,
2. $\mathsf{plus}fg \circ \mathrm{inr} = \mathrm{inr} \circ g$,
3. $\mathsf{plus}fg \circ \mathsf{plus}hk = \mathsf{plus}(f \circ h)(g \circ k)$,
4. $\mathsf{times}fg \circ \mathsf{times}hk = \mathsf{times}(f \circ h)(g \circ k)$,
5. $[f, g] \circ \mathsf{plus}hk = [f \circ h, g \circ k]$,
6. $\mathsf{plus}hk \circ <f, g> = <h \circ f, k \circ g>$.

(In general we don't have $\mathrm{fst} \circ \mathsf{times}fg = f \circ \mathrm{fst}$ or $\mathrm{snd} \circ \mathsf{times}fg = g \circ \mathrm{snd}$.)

Positive (negative) type schemes can really be viewed as (contravariant) functors in the syntax of polymorphic lambda calculus. (Consider a syntax with countably many variables of every type and view types as objects and terms of type $\sigma\to\tau$ as morphisms from σ to τ.) The positive type schemes are a syntactic version of covariant functors. Similarly we also have syntactic versions of weakly initial (terminal) (co)algebras and (co)recursive (co)algebras.

Definition 4.5 *Suppose we work in (an extension of) polymorphic lambda calculus where we have fixed a notation for weak products and coproducts (e.g. the second order definable ones.) Let $\Phi(\alpha)$ be a positive type scheme.*

1. *The triple $(\sigma_0, M_0, \mathrm{Elim})$ is a* syntactic weakly initial Φ-algebra *if*
   (a) $\sigma_0 \in \mathbf{T}$,
   (b) $\vdash M_0{:}\Phi(\sigma_0)\to\sigma_0$,

(c) $\vdash \mathrm{Elim}{:}\forall\beta.(\Phi(\beta)\to\beta)\to\sigma_0\to\beta$,

such that

$$\mathrm{Elim}\tau g \circ M_0 = g \circ \Phi(\mathrm{Elim}\tau g)$$

for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\Phi(\tau)\to\tau$.

2. *The triple* $(\sigma_1, M_1, \mathrm{Intro})$ *is a* syntactic weakly terminal Φ-coalgebra *if*

   (a) $\sigma_1 \in \mathbf{T}$,

   (b) $\vdash f_1{:}\sigma_1\to\Phi(\sigma_1)$,

   (c) $\vdash \mathrm{Intro}{:}\forall\beta.(\beta\to\Phi(\beta))\to\beta\to\sigma_1$,

   such that

$$M_1 \circ \mathrm{Intro}\tau g = \Phi(\mathrm{Intro}\tau g) \circ g$$

   for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\tau\to\Phi(\tau)$.

3. *The triple* $(\sigma_0, M_0, \mathrm{Rec})$ *is a* syntactic recursive Φ-algebra *if*

   (a) $\sigma_0 \in \mathbf{T}$,

   (b) $\vdash M_0{:}\Phi(\sigma_0)\to\sigma_0$,

   (c) $\vdash \mathrm{Rec}{:}\forall\beta.(\Phi(\beta\times\sigma_0)\to\beta)\to\sigma_0\to\beta$,

   such that

$$\mathrm{Rec}\tau g \circ M_0 = g \circ \Phi(<\mathrm{Rec}\tau g, \mathrm{id}>)$$

   for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\Phi(\tau\times\sigma_0)\to\tau$.

4. *The triple* $(\sigma_1, M_1, \mathrm{Corec})$ *is a* syntactic corecursive Φ-coalgebra *if*

   (a) $\sigma_1 \in \mathbf{T}$,

   (b) $\vdash f_1{:}\sigma_1\to\Phi(\sigma_1)$,

   (c) $\vdash \mathrm{Corec}{:}\forall\beta.(\beta\to\Phi(\beta+\sigma_1))\to\beta\to\sigma_1$,

   such that

$$M_1 \circ \mathrm{Corec}\tau g = \Phi([\mathrm{Corec}\tau g, \mathrm{id}]) \circ g$$

   for any $\tau \in \mathbf{T}$ *and* $\Gamma \vdash g{:}\tau\to\Phi(\tau+\sigma_1)$.

We have the following proposition, of which the first part is a syntactic version of a result in [Reynolds and Plotkin 1990] and the second part is a result of [Wraith 1989]. In fact, the first part of the proposition says that the algebraic inductive data types can be represented in F, which result originally goes back to [Böhm and Berarducci 1985]. Here we just want to give these representations in short; for further details one may consult [Böhm and Berarducci 1985], [Leivant 1989] or [Girard et al. 1989].

Proposition 4.6 *We work in the system F. Let $\Phi(\alpha)$ be a positive type scheme. Then*

1. *There is a syntactic weakly initial Φ-algebra.*

2. *There is a syntactic weakly terminal Φ-coalgebra.*

Proof Let $\Phi(\alpha)$ be a positive type scheme.

1. Define $\sigma_0 := \forall\alpha.(\Phi(\alpha)\to\alpha)\to\alpha$, $M_0 := \lambda x{:}\Phi(\sigma).\lambda\alpha.\lambda g{:}\Phi(\alpha)\to\alpha.g(\Phi(\text{Elim}\alpha g)x)$, and Elim $:= \lambda\alpha.\lambda g{:}\Phi(\alpha)\to\alpha.\lambda y{:}\sigma.y\alpha g$. Now $(\sigma_0, M_0, \text{Elim})$ is a syntactic weakly initial Φ-algebra.

2. Define $\sigma_1 := \forall\alpha.(\forall\beta.(\beta\to\Phi(\beta))\to\beta\to\alpha)\to\alpha$,
$M_1 := \lambda x{:}\sigma.x(\Phi(\sigma))(\lambda\beta.\lambda g{:}\beta\to\Phi(\beta).\lambda z{:}\beta.\Phi(\text{Intro}\beta g)(gx))$, and
Intro $:= \lambda\alpha.\lambda g{:}\alpha\to\Phi(\alpha).\lambda y{:}\alpha.\lambda\beta.\lambda h{:}\forall\gamma.(\gamma\to\Phi(\gamma))\to\gamma\to\beta.h\alpha g y$. Now $(\sigma_1, M_1, \text{Intro})$ is a syntactic weakly terminal Φ-coalgebra.

We don't know whether there are syntactic recursive algebras or syntactic corecursive coalgebras in F. The answer seems to be negative. The well-known definitions of algebraic data-types in F (which are almost the ones defined in the proof above) do in general not allow recursion or corecursion, as will be illustrated by looking at the examples of natural numbers and streams of natural numbers. This means that recursion and corecursion have to be defined in terms of iteration and coiteration, using the techniques discussed in the Examples 2.7 and 2.8. As was noticed there, it makes a difference whether product and coproduct are weak or semi, so let's note the following fact.

Fact 4.7 *The definable coproduct in F is a weak coproduct, but the definable product in F is a semi product.*
(That is, $<f,g>\circ h = <f\circ h, g\circ h>$, but not $h\circ[f,g] = [h\circ f, h\circ g]$)

Example 4.8 *(See also Example 3.5 and Proposition 3.6.) We define recursive functions on the weak initial algebra of natural numbers.*
Let $(\text{Nat}, M_0, \text{Elim})$ *be the syntactic weak initial algebra of* $\Phi(\alpha) = 1+\alpha$, *as given in the proof of 4.6, where* 1 *and* $+$ *are the second order definable ones. (One can also take the well-known polymorphic Church numerals, which is a slight modification of our type* Nat. *The exposition is not essentially different, but we want to use our categorical understanding of recursion of 2.7.)*
So $\text{Nat} = \forall\alpha((1+\alpha)\to\alpha)\to\alpha$, $M_0 = \lambda x{:}1+\text{Nat}.\lambda\alpha.\lambda g.g((\text{id}+\text{Elim}\alpha g)x)$ *and*
$\text{Elim} = \lambda\alpha.\lambda g.\lambda y{:}\text{Nat}.y\alpha g$. *Now we first define* $\mathbf{Z} := M_0\circ\text{inl}$ *and* $\mathbf{S} := M_0\circ\text{inr}$.
Following Example 2.7, we now define $\text{Rec}g = \text{fst}\circ\text{Elim}(\tau\times\text{Nat})(<g,[\mathbf{Z},\mathbf{S}\circ\text{snd}]>)$, *for* $g{:}1+\tau\times\text{Nat}\to\tau$. *If*

$$g = [g_1, k\circ<\text{fst},\text{snd}>]$$

for some $k{:}\tau\times\text{Nat}\to\tau$, *we obtain the recursion equalities for* $\text{Rec}g$*:*

$$\begin{aligned}\text{Rec}g\circ\mathbf{Z} &= g_1,\\ \text{Rec}g\circ\mathbf{S}^{n+1}\circ\mathbf{Z} &= k\circ<\text{Rec}g,\text{id}>\circ\mathbf{S}^n\circ\mathbf{Z}.\end{aligned}$$

(See 2.7 for the restriction on the form of g; the product is semi here.) The predecessor is now defined by taking $g = [\mathbf{Z},\text{snd}]$, *so* $P := \text{fst}\circ\text{Elim}(\tau\times\text{Nat})(<[\mathbf{Z},\text{snd}],[\mathbf{Z},\mathbf{S}\circ\text{snd}]>)$. *Notice that* $P(\mathbf{S}t) = t$ *only for standard natural numbers, i.e. for* $t = \mathbf{S}^n(\mathbf{Z}*)$, *with* $*$ *the unique (closed) term of type* 1. *Also notice that P computes the predecessor of a natural number n in a number of steps of order n.*

Example 4.9 *We define corecursive functions on streams of natural numbers. Take for* Stream *the syntactic weakly terminal Φ-coalgebra as in the proof of 4.6, for* $\Phi(\alpha) = \text{Nat}\times\alpha$. *So* $\text{Stream} = \forall\alpha.(\forall\beta.(\beta\to(\text{Nat}\times\beta))\to\beta\to\alpha)\to\alpha$, $M_1 = \lambda x{:}\text{Stream}.x(\text{Nat}\times\sigma)(\lambda\beta.\lambda g{:}\beta\to\text{Nat}\times$

$\beta.\lambda z{:}\beta.(\mathrm{id}\times \mathrm{Intro}\beta g)(gx))$, *and*
$\mathrm{Intro} = \lambda\alpha.\lambda g{:}\alpha\to\mathrm{Nat}\times\alpha.\lambda y{:}\alpha.\lambda\beta.\lambda h{:}\forall\gamma.(\gamma\to\mathrm{Nat}\times\gamma)\to\gamma\to\beta.h\alpha g y$. *We can define head and tail functions by taking* $\mathbf{H} := \mathrm{fst}\circ M_1$ *and* $\mathbf{T} := \mathrm{snd}\circ M_1$. *Following Example 2.8, we now define for* $g{:}\tau\to\mathrm{Nat}\times(\tau+\mathrm{Stream})$ $\mathrm{Corec}g := \mathrm{Intro}(\tau+\mathrm{Stream})([g,<H,\mathrm{inr}\circ\mathbf{T}>])\circ\mathrm{inl}$. *As the coproduct is not semi, but weak (see 2.8), we find that only for* $g = <g_1, \mathrm{in}\circ k>$ *for* in *is* inr *or* inl *and some* $k{:}\mathrm{Stream}\to B$ *or* $k{:}\mathrm{Stream}\to\mathrm{Stream}$ *we obtain the corecursion equations.*

$$\begin{aligned}\mathbf{H}\circ\mathrm{Corec}g &= g_1,\\ \mathbf{T}\circ\mathrm{Corec}g &= [\mathrm{Corec}g,\mathrm{id}]\circ\mathrm{snd}\circ g.\end{aligned}$$

The function that replaces the head of a stream by zero is now defined by $\mathrm{ZeroH} := \mathrm{Corec}<\mathbf{Z},\mathrm{inr}\circ\mathbf{T}>$

It is really impossible to define a 'global' predecessor on the weakly initial natural numbers as described above (and similarly for the polymorphic Church numerals.) Also it is impossible to define a global ZeroH-function on the weakly terminal streams as described above. This is shown in the following proposition.

Proposition 4.10 *1. For* $\mathrm{Nat} = \forall\alpha((1+\alpha)\to\alpha)\to\alpha$, *there is no closed term* $P{:}\mathrm{Nat}\to\mathrm{Nat}$ *such that* $P(\mathbf{S}x) = x$ *for* x *a variable.*

2. For $\mathrm{Stream} := \forall\beta.(\forall\gamma.(\gamma\to\mathrm{Nat})\to(\gamma\to\gamma)\to\gamma\to\beta)\to\beta$ *there is no closed term* $\mathrm{ZeroH}{:}\mathrm{Stream}\to\mathrm{Stream}$ *such that* $\mathbf{T}(\mathrm{ZeroH}y) = \mathbf{T}y$ *and* $\mathbf{H}(\mathrm{ZeroH}y) = 0$ *for* y *a variable.*

Proof Both cases immediately by the Church-Rosser property for the system F.

One can show in general that the (co)inductive types in system F as defined above do not allow (co)recursion, i.e. they are weakly initial (terminal) (co)algebras.

5 Recursive algebras and corecursive coalgebras and polymorphism

We define an extension of F which includes a syntactic formalization of recursive algebras and corecursive coalgebras. Then we show the remarkable fact that in this system one can define recursive algebras in terms of corecursive coalgebras and vice versa, so one of the two is enough to be able to define the other. This fact has a counterpart in semantics in the form that every K-model of polymorphic lambda calculus that has a recursive T-algebras for every expressible functor T, also has a corecursive T-coalgebra for every expressible functor T and vice versa. (The notion of K-model is in [Reynolds and Plotkin 1990]; it is a syntax dependent notion of model for F, described by giving a set of constraints that a structure and an interpretation function should satisfy in order to be a model. As it covers a lot of known models it serves well as a framework for stating this property semantically. Also the notion of expressible functor comes from [Reynolds and Plotkin 1990]; roughly speaking, a functor is expressible if there is a type scheme whose interpretation in the model (as a function of the free type variable) is a the functor.)
We then want to relate our extension of F with recursive (and corecursive) types to a system described by [Mendler 1987]. The latter system has a different scheme for recursive (and corecursive) types, the syntax of which is a bit too weak to define one in terms of the other.

We can, however, interpret our system with recursive and corecursive types in Mendler's (with either recursive or corecursive types.) This is done by showing that the system has syntactic recursive Φ-algebras and syntactic corecursive Φ-coalgebras for every positive type scheme Φ. (See Definition 4.5.)

Definition 5.1 *The system $F_{(co)rec}$ is the system F extended with the following.*

1. *The set of types* $\mathbf{T}$ *is extended with* $\mu\alpha.\Phi(\alpha)$ *and* $\nu\alpha.\Phi(\alpha)$*, for* $\Phi(\alpha)$ *a positive type scheme.*
2. *For* $\mu\alpha.\Phi(\alpha)$ *and* $\nu\alpha.\Phi(\alpha)$ *we have the extra constants*

$$\begin{array}{rl} \mathbf{In}_\mu : \Phi(\mu)\to\mu & \mathrm{Rec}_\mu : \forall\alpha.(\Phi(\alpha\times\mu)\to\alpha)\to\mu\to\alpha, \\ \mathbf{Out}_\nu : \nu\to\Phi(\nu) & \mathrm{Corec}_\nu : \forall\alpha.(\alpha\to\Phi(\alpha+\nu))\to\alpha\to\nu. \end{array}$$

3. *Reduction rules for* μ *and* ν*:*

$$\begin{array}{rcl} \mathrm{Rec}\tau g(\mathbf{In}x) & \longrightarrow_\mu & g(\Phi(<\mathrm{Rec}\tau g, \mathrm{id}>)x), \\ \mathbf{Out}(\mathrm{Corec}\tau g x) & \longrightarrow_\nu & \Phi([\mathrm{Corec}\tau g, \mathrm{id}])(gx). \end{array}$$

(μ abbreviates $\mu\alpha.\Phi(\alpha)$ and ν abbreviates $\nu\alpha.\Phi(\alpha)$.)

We now have the following theorem, stating that in polymorphic lambda calculus, if one has recursive types the corecursive types can be defined and vice versa.

Theorem 5.2 *In F we can define ν,* **Out** *and* Corec *in terms of μ,* **In** *and* Rec *and vice versa.*

Proof Suppose we only have the rules for μ, **In** and Rec and let Φ be a positive type scheme. Define

$$\begin{array}{rcl} \Theta(\alpha) & := & \forall\gamma.(\forall\beta.(\beta\to\Phi(\beta+\alpha)\times\beta)\to\gamma)\to\gamma, \\ \sigma & := & \mu\alpha.\Theta(\alpha), \\ \mathrm{Corec} & := & \lambda\alpha.\lambda g{:}\alpha\to\Phi(\alpha+\sigma).\lambda x{:}\alpha.\mathbf{In}_\sigma(\lambda\gamma.\lambda h{:}\forall\beta.(\beta\to\Phi(\beta+\sigma)\times\beta)\to\gamma.h\alpha<g,x>), \\ \mathbf{Out} & := & \mathrm{Rec}_\sigma(\Phi(\sigma))(\lambda z{:}\Theta(\Phi(\sigma)\times\sigma).z(\Phi(\sigma))(\lambda\gamma.\lambda p.\Phi([\mathrm{Corec}\gamma(\Phi(\mathsf{plus}(\mathrm{id})(\mathrm{snd}))\circ p_1),\mathrm{snd}])(p_1p_2))), \end{array}$$

where p_1 and p_2 abbreviate $\mathrm{fst}p$ and $\mathrm{snd}p$ (the type of p is $(\gamma\to\Phi(\gamma+(\Phi(\sigma)\times\sigma)))\times\sigma$.) We have

$$\mathrm{Rec}_\sigma\tau g(\mathbf{In}_\sigma x) \longrightarrow_\mu g(\Theta(<\mathrm{Rec}_\sigma\tau g, \mathrm{id}>)x),$$

for any g and x of appropriate types and we want to show

$$\mathbf{Out}(\mathrm{Corec}\tau g x) \twoheadrightarrow \Phi([\mathrm{Corec}\tau g, \mathrm{id}])(gx))$$

for $g{:}\tau\to\Phi(\tau+\sigma)$ and $x{:}\tau$.

To make things easier to read we omit the type information in lambda abstractions. Let τ be a type, $g{:}\tau\to\Phi(\tau+\sigma)$ and $x{:}\tau$ and abbreviate $F \equiv \lambda\gamma p.\Phi([\mathrm{Corec}\gamma(\Phi(\mathsf{plus}(\mathrm{id})(\mathrm{snd}))\circ p_1),\mathrm{snd}])(p_1p_2)$. The lifting of an f via Θ is defined as (following 4.2)

$$\Theta(f) \equiv \lambda t\gamma k.t\gamma(\lambda\beta z.k\beta(\mathsf{times}(\lambda yx.\mathsf{plus}(\mathrm{id})f(yx))(\mathrm{id})z).$$

Now

$$\begin{array}{rcl}\mathbf{Out}(\mathrm{Corec}\tau g x) & \twoheadrightarrow & \mathrm{Rec}_\sigma(\Phi(\sigma))(\lambda z.z(\Phi(\sigma))F)(\mathbf{In}_\sigma(\lambda\gamma h.h\tau{<}g,x{>})) \\ & \twoheadrightarrow & \Theta({<}\mathrm{Rec}_\sigma(\Phi(\sigma))(\lambda z.z(\Phi(\sigma))F),\mathrm{id}{>})(\lambda\gamma h.h\tau{<}g,x{>})(\Phi(\sigma))F \\ & \twoheadrightarrow & (\lambda\beta z.F\beta(\mathsf{times}(\lambda yx.\Phi(\mathsf{plus}(\mathrm{id})F)(yx)(\mathrm{id})z))\tau{<}g,x{>} \\ & \twoheadrightarrow & \Phi([\mathrm{Corec}\tau(\Phi(\mathsf{plus}(\mathrm{id})(\mathrm{snd}))\circ\Phi(\mathsf{plus}(\mathrm{id})F)\circ g,\mathrm{snd}])(\Phi(\mathsf{plus}(\mathrm{id})F)(gx)) \\ & \twoheadrightarrow & \Phi([\mathrm{Corec}\tau g,\mathrm{snd}]\circ\mathsf{plus}(\mathrm{id})F))(gx) \\ & \twoheadrightarrow & \Phi([\mathrm{Corec}\tau g,\mathrm{id}])(gx).\end{array}$$

The other way around, suppose we only have rules for ν, **Out** and Corec and let Φ be a positive type scheme. Define

$$\begin{array}{rcl}\Theta(\alpha) & = & \forall\beta.(\Phi(\beta\times\alpha)\to\beta)\to\beta, \\ \sigma & := & \nu\alpha.\Theta(\alpha), \\ \mathrm{Rec} & := & \lambda\alpha\lambda g{:}\Phi(\alpha\times\sigma)\to\alpha.\lambda x{:}\sigma.\mathbf{Out}_\sigma x\alpha g, \\ \mathbf{In} & := & \mathrm{Corec}_\sigma(\Phi(\sigma))(\lambda z{:}\Phi(\sigma).\lambda\beta.\lambda h{:}\Phi(\beta\times\Phi(\sigma))\to\beta.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})z)).\end{array}$$

We have

$$\mathbf{Out}_\sigma(\mathrm{Corec}_\sigma\tau g x)\twoheadrightarrow\Theta([\mathrm{Corec}_\sigma\tau g,\mathrm{id}])(gx))$$

and we want to prove

$$\mathrm{Rec}\tau g(\mathbf{In}x)\twoheadrightarrow g(\Phi({<}\mathrm{Rec}\tau g,\mathrm{id}{>})x).$$

We omit again type information in lambda abstractions and abbreviate $F\equiv\lambda z\beta h.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})z)$. According to Definition 4.2,

$$\Theta(f)\equiv\lambda t\beta k.t\beta(\lambda z.k(\Phi(\mathsf{times}(\mathrm{id})f)z)).$$

Now

$$\begin{array}{rcl}\mathrm{Rec}\tau g(\mathbf{In}x) & \twoheadrightarrow & \mathbf{Out}_\sigma(\mathrm{Corec}_\sigma(\Phi(\sigma))Fx)\tau g \\ & \twoheadrightarrow & \Theta([\mathrm{Corec}_\sigma(\Phi(\sigma))F,\mathrm{id}])(\lambda\beta h.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})x)\tau g \\ & \twoheadrightarrow & (\lambda\beta h.h(\Phi({<}\mathrm{Rec}\beta(h\circ\Phi(\mathrm{id}\times\mathrm{inr})),\mathrm{inr}{>})x)\tau(\lambda z.g(\Phi(\mathsf{times}(\mathrm{id})F)z)) \\ & \twoheadrightarrow & g(\Phi(\mathsf{times}(\mathrm{id})F)(\Phi({<}\mathrm{Rec}\tau(\lambda z.g(\Phi(\mathsf{times}(\mathrm{id})F)z)\circ\Phi(\mathsf{times}(\mathrm{id})(\mathrm{inr})),\mathrm{inr}{>})x) \\ & \twoheadrightarrow & g(\Phi({<}\mathrm{Rec}\tau(g\circ\Phi(\mathsf{times}(\mathrm{id})F)\circ\Phi(\mathsf{times}(\mathrm{id})(\mathrm{inr}),\mathrm{id}{>})x) \\ & \twoheadrightarrow & g(\Phi({<}\mathrm{Rec}\tau g,\mathrm{id}{>})x)\end{array}$$

We now want to look at the system of recursive types, as defined by [Mendler 1987], let's call it $F_{(CO)REC}$. (The system also has corecursive types.)

Definition 5.3 ([Mendler 1987]) *The system $F_{(CO)REC}$ is defined by adding to the polymorphic lambda calculus the following.*

1. *The set of types* $\mathbf{T}$ *is extended with $\mu\alpha.\Phi(\alpha)$ and $\nu\alpha.\Phi(\alpha)$, for $\Phi(\alpha)$ a positive type scheme.*
2. *For $\mu\alpha.\Phi(\alpha)$ and $\nu\alpha.\Phi(\alpha)$ we have the extra constants*

$$\begin{array}{ll}\mathbf{in}_\mu:\Phi(\mu)\to\mu & \mathrm{R}_\mu:\forall\beta.(\forall\gamma.(\gamma\to\mu)\to(\gamma\to\beta)\to\Phi(\gamma)\to\beta)\to\mu\to\beta, \\ \mathbf{out}_\nu:\nu\to\Phi(\nu) & \Omega_\nu:\forall\beta.(\forall\gamma.(\nu\to\gamma)\to(\beta\to\gamma)\to\beta\to\Phi(\gamma))\to\beta\to\nu.\end{array}$$

3. Reduction rules for μ and ν:

$$\begin{array}{rcl} \mathrm{R}_\mu \tau g(\mathbf{in}_\mu x) & \longrightarrow_\mu & g\mu(\mathrm{id}_\mu)(\mathrm{R}_\mu \tau g)x, \\ \mathbf{out}_\nu(\Omega_\nu \tau g x) & \longrightarrow_\nu & g\nu(\mathrm{id}_\nu)(\Omega_\nu \tau g)x. \end{array}$$

(μ abbreviates $\mu\alpha.\Phi(\alpha)$ and ν abbreviates $\nu\alpha.\Phi(\alpha)$.)

In [Mendler 1987] it is shown that this system satisfies a lot of nice meta-properties, like strong normalization and confluence of the reduction relation.

Definition 5.4 *The system $F_{(CO)REC}$ with only the rules for μ will be called F_{REC} and similarly, $F_{(CO)REC}$ with only the rules for ν will be called F_{COREC}.*

We show that the system $F_{(co)rec}$ can be defined in both F_{REC} and F_{COREC}, so both systems have all syntactic recursive algebras and all syntactic corecursive coalgebras.

Proposition 5.5 *1. The μ-types of $F_{(co)rec}$ can be defined in F_{REC}.*

2. The ν-types of $F_{(co)rec}$ can be defined in F_{COREC}.

Proof Let $\Phi(\alpha)$ be a positive type scheme.

1. Write μ for $\mu\alpha.\Phi(\alpha)$ and take $\mathbf{In} = \mathbf{in}$,
   $\mathrm{Rec}_\mu = \lambda\beta.\lambda g{:}\Phi(\beta\times\mu)\rightarrow\beta.\mathrm{R}_\mu\beta(\lambda\alpha.\lambda f{:}\alpha\rightarrow\mu.\lambda k{:}\alpha\rightarrow\beta.g\circ\Phi(<k,f>))$. Then μ, $\mathbf{In}$ and Rec_μ together define the μ-type of $F_{(co)rec}$ in the sense that $\mathrm{Rec}_\mu \tau g(\mathbf{In}x) = g(\Phi(<\mathrm{Rec}_\mu\tau g, \mathrm{id}>)x)$.

2. Write ν for $\nu\alpha.\Phi(\alpha)$ and take $\mathbf{Out} = \mathbf{out}$,
   $\mathrm{Corec}_\nu = \lambda\beta.\lambda g{:}\beta\rightarrow\Phi(\beta+\nu).\Omega\beta(\lambda\alpha.\lambda f{:}\nu\rightarrow\alpha.\lambda k{:}\beta\rightarrow\alpha.\Phi([k,f])\circ g)$. Then ν, $\mathbf{Out}$ and Corec_ν together define the ν-type of $F_{(co)rec}$ because $\mathbf{Out}(\mathrm{Corec}_\nu\tau g x) = \Phi([\mathrm{Corec}_\nu\tau g, \mathrm{id}])(gx)$.

Corollary 5.6 *1. $F_{(co)rec}$ can be defined in both F_{REC} and F_{COREC}.*

2. For every positive type scheme $\Phi(\alpha)$ both F_{REC} and F_{COREC} have a syntactic recursive Φ-algebra and a syntactic corecursive Φ-coalgebra.

Proof Both immediately by the proposition and Theorem 5.2.

As a corollary of the translation of $F_{(co)rec}$ in to $F_{(CO)REC}$ we find that $F_{(co)rec}$ is strongly normalizing.

Proposition 5.7 *The reduction relation of the system $F_{(co)rec}$ is strongly normalizing and confluent.*

Proof In order to prove strong normalization we define a mapping $[-]$ from the terms of $F_{(co)rec}$ to the term of $F_{(CO)REC}$ that preserves infinite reduction paths. Then $F_{(co)rec}$ is strongly normalizing by the fact that $F_{(CO)REC}$ is strongly normalizing (see [Mendler 1987].) One easily verifies that the system is weakly confluent (i.e. if $M \longrightarrow N$ and $M \longrightarrow$P, then $\exists Q[N \twoheadrightarrow Q \& P \twoheadrightarrow Q]$.) The confluence then follows from Newman's lemma ([Newman 1942]), stating that strong normalization and weak confluence together imply confluence.

The definition of $[-]$ is very similar to the mapping defined in the proof of Proposition 5.5. (As the types do not in any way interfere with the reduction process we omit the types in abstractions.) Define $[-]$ by

$$\begin{array}{rcl}
[\mathrm{Rec}_\mu] & := & \lambda\beta g.\mathrm{R}_\mu\beta(\lambda\alpha fk.g\circ\Phi(<k,f>)),\\
[\mathrm{Rec}_\mu\tau] & := & \lambda g.\mathrm{R}_\mu\tau(\lambda\alpha fk.g\circ\Phi(<k,f>)),\\
[\mathrm{Rec}_\mu\tau g] & := & \mathrm{R}_\mu\tau(\lambda\alpha fk.g\circ\Phi(<k,f>)),\\
[\mathbf{In}] & := & \mathbf{in},\\
[\mathrm{Corec}_\nu] & := & \lambda\beta g.\Omega\beta(\lambda\alpha fk.\Phi([k,f])\circ g),\\
[\mathrm{Corec}_\nu\tau] & := & \lambda g.\Omega\tau(\lambda\alpha fk.\Phi([k,f])\circ g),\\
[\mathrm{Corec}_\nu\tau g] & := & \Omega\tau(\lambda\alpha fk.\Phi([k,f])\circ g),\\
[\mathbf{Out}] & := & \mathbf{out},
\end{array}$$

and further by induction on the structure of the terms. Then

$$\begin{array}{ll}
M\longrightarrow_\beta N & \Rightarrow [M]\twoheadrightarrow^+_\beta [N],\\
M\longrightarrow_\eta N & \Rightarrow [M]\twoheadrightarrow_\eta [N],\\
M\longrightarrow_\mu N & \Rightarrow [M]\longrightarrow_\mu [N],\\
M\longrightarrow_\nu N & \Rightarrow [M]\longrightarrow_\nu [N],
\end{array}$$

where $\twoheadrightarrow^+_\beta$ denotes a reduction in at least one step. As there is no infinite η-reduction in $F_{(co)rec}$, the mapping $[-]$ maps an infinite reduction path in $F_{(co)rec}$ to an infinite reduction path in $F_{(CO)REC}$, so we are done.

It doesn't seem possible to define the μ-types in terms of the ν-types in $F_{(CO)REC}$, nor to define the system F_{COREC} in the system F_{corec}. When one attempts to do so, some extra equalities seem to be required.

6 Discussion

As pointed out by Christine Paulin ([Paulin 1992]) the technique in the proof of Theorem 5.2 also applies to a polymorphic lambda calculus with a kind of 'retract types' that we shall describe now. We give the syntax as it has been communicated to us by Christine Paulin; it is implicit in papers by Parigot ([Parigot 1988] and [Parigot 1992]), where extensions of the system AF2 with recursive types are studied. (AF2 is a system of second oder predicate logic with an interpretation of proofs as untyped lambda terms.) The connections between our system with recursive types and the (extensions) of AF2 is a subject which needs further investigation; we feel that this is not the place to do so.

Definition 6.1 *The system F_{ret} is the extension of system F with the following.*

1. *The set of types $\mathbf{T}$ is extended with $\rho\alpha.\Phi(\alpha)$, for $\Phi(\alpha)$ a positive type scheme.*
2. *For $\rho\alpha.\Phi(\alpha)$ we have the extra constants*

$$\mathbf{i}_\rho:\Phi(\rho)\to\rho,\mathbf{o}_\rho:\rho\to\Phi(\rho).$$

3. Reduction rule for ρ:

$$\mathbf{o}_\rho(\mathbf{i}_\rho x) \longrightarrow_\rho x.$$

(ρ abbreviates $\rho\alpha.\Phi(\alpha)$.)

In this system one can construct for $\Phi(\alpha)$ a positive type scheme a type ρ with $\Phi(\rho) < \rho$ ($\Phi(\rho)$ is a retract of ρ.) As pointed out to us by [Paulin 1992], the technique of 5.2 can be applied to obtain that both the systems F_{rec} and F_{corec} can be defined in F_{ret}. Also the reverse holds: F_{ret} can be defined in both F_{rec} and F_{corec}. It would be an interesting subject for further investigations to see how the retract types relate to the recursive and corecursive types on the categorical level.

Theorem 6.2 *The systems F_{rec} and F_{corec} can be defined in F_{ret} and vice versa.*

Proof To define F_{rec} in F_{ret} take

$$\begin{array}{rcl}
\mu\alpha.\Phi(\alpha) & := & \rho\alpha.\forall\gamma.(\Phi(\gamma\times\alpha)\to\alpha)\to\alpha, \\
\mathrm{Rec}_\mu & := & \lambda\gamma.\lambda f{:}\Phi(\gamma\times\mu)\to\gamma.\lambda x{:}\mu.\mathbf{o}x\gamma f, \\
\mathbf{In}_\mu & := & \lambda x{:}\Phi(\mu).\mathbf{i}(\lambda\gamma.\lambda f{:}\Phi(\gamma\times\mu)\to\mu.f(\Phi(<\mathrm{Rec}_\mu\gamma f,\mathrm{id}>))x)),
\end{array}$$

and $\mathrm{Rec}_\mu\tau g(\mathbf{In}_\mu x) \twoheadrightarrow g(\Phi(<\mathrm{Rec}_\mu\tau g,\mathrm{id}>)x)$ easily follows. To define F_{corec} in F_{ret} take

$$\begin{array}{rcl}
\nu\alpha.\Phi(\alpha) & := & \rho\alpha.\exists\gamma.(\gamma\to\Phi(\gamma+\alpha))\to\gamma), \\
\mathrm{Corec}_\nu & := & \lambda\gamma.\lambda f{:}\gamma\to\Phi(\gamma+\nu).\lambda x{:}\gamma.\mathbf{i}(\lambda\beta\lambda k.k\gamma<f,x>), \\
\mathbf{Out}_\nu & := & \lambda x{:}\nu.\mathbf{o}x(\Phi(\nu))(\lambda\gamma.\lambda f{:}\gamma\to\Phi(\gamma+\nu)\times\nu.\Phi([\mathrm{Corec}_\nu\gamma(\mathrm{fst}f),\mathrm{id}])(\mathrm{fst}f(\mathrm{snd}f))),
\end{array}$$

and $\mathbf{Out}_\nu(\mathrm{Corec}_\nu\tau g) \twoheadrightarrow \Phi([\mathrm{Corec}_\nu\tau g,\mathrm{id}])(gx)$ easily follows.

To define F_{ret} in terms of F_{rec} or F_{corec}, take respectively

$$\begin{array}{rcl}
\rho\alpha.\Phi(\alpha) & := & \mu\alpha.\Phi(\alpha), \\
\mathbf{i} & := & \mathbf{In}_\mu, \\
\mathbf{o} & := & \mathrm{Rec}_\mu\Phi(\mu)(\Phi(\mathrm{snd}))
\end{array}$$

(so $\mathbf{o}(\mathbf{i}x) \twoheadrightarrow x$) and

$$\begin{array}{rcl}
\rho\alpha.\Phi(\alpha) & := & \nu\alpha.\Phi(\alpha), \\
\mathbf{o} & := & \mathbf{Out}_\nu, \\
\mathbf{i} & := & \mathrm{Corec}_\mu\Phi(\nu)(\Phi(\mathrm{inr}))
\end{array}$$

(so again, $\mathbf{o}(\mathbf{i}x) \twoheadrightarrow x$.)

We can collect the results from Theorems 5.2 and 6.2 and Corollary 5.6 in a picture as follows. (An arrow from A to B means that the system A can be translated in the system B.)

$$\begin{tikzcd}
 & F_{ret} \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & \\
F_{rec} \arrow[rr, leftrightarrow] \arrow[d] & & F_{corec} \arrow[d] \\
F_{REC} & & F_{COREC}
\end{tikzcd}$$

If we translate in F_{rec} the type $\mu\alpha.\Phi(\alpha)$ in terms of the ρ-type, which is defined in terms of the μ-type, we obtain the type $\mu\alpha.\forall\gamma.(\Phi(\gamma \times \alpha) \to \gamma) \to \gamma$. A similar situation occurs if we translate in F_{ret} a ρ-type in terms of a μ-type, which is defined in terms of the ρ-type: $\rho\alpha.\Phi(\alpha)$ becomes $\rho\alpha.\forall\gamma.(\Phi(\gamma \times \alpha) \to \gamma) \to \gamma$. Using these double translations, we can deduce the following facts about the systems F_{rec}, F_{corec} and F_{ret} themselves.

Fact 6.3 *1. For ρ a retract type of $\Psi(\alpha) \equiv \forall\gamma.(\Phi(\gamma \times \alpha) \to \gamma) \to \gamma$ or of $\Psi(\alpha) \equiv \exists\gamma.(\gamma \to \Phi(\gamma + \alpha)) \times \gamma$, ρ is also a retract type of Φ.*

2. For μ a recursive type of $\Psi(\alpha) \equiv \forall\gamma.(\Phi(\gamma \times \alpha) \to \gamma) \to \gamma$, μ is also a recursive type of Φ.

3. For ν a corecursive type of $\Psi(\alpha) \equiv \exists\gamma.(\gamma \to \Phi(\gamma + \alpha)) \times \gamma$, ν is also a corecursive type of Φ.

We can also compose the translations to obtain new interpretations of μ-types in ν-types and vice versa:

Fact 6.4 *1. For $\Phi(\alpha)$ a positive type scheme, we can interpret ν-types in F_{rec} by taking $\nu\alpha.\Phi(\alpha)$ and Corec_ν as in 5.2 and*

$$\mathbf{Out}_\nu :\equiv \lambda x.\mathrm{Rec}_\nu(\Theta(\nu))(\Theta(\mathrm{snd}))x(\Phi(\nu))(\lambda\gamma f.\Phi([\mathrm{Corec}\gamma(\mathrm{fst} f), \mathrm{id}])(\mathrm{fst} f(\mathrm{snd} f))).$$

2. For $\Phi(\alpha)$ a positive type scheme, we can interpret μ-types in F_{corec} by taking $\mu\alpha.\Phi(\alpha)$ and Rec_μ as in 5.2 and

$$\mathbf{In}_\mu :\equiv \lambda x.\mathrm{Corec}_\mu(\Theta(\mu))(\Theta(\mathrm{inr}))(\lambda\gamma f.f(\Phi(<\mathrm{Rec}\gamma f, \mathrm{id}>)x)).$$

References

[Böhm and Berarducci 1985] C. Böhm and A. Berarducci, Automatic synthesis of typed Λ-programs on term algebras *Theor. Comput. Science*, 39, pp 135-154.

[Coquand and Huet 1988] Th. Coquand and G. Huet, The calculus of constructions, *Information and Computation*, 76, pp 95-120.

[Coquand and Mohring 1990] Inductively defined types, In P. Martin-Löf and G. Mints editors. *COLOG-88 : International conference on computer logic, LNCS 417.*

[Dowek e.a. 1991] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, B. Werner, The Coq proof assistant version 5.6, user's guide. INRIA Rocquencourt - CNRS ENS Lyon.

[Girard et al. 1989] J.Y. Girard, Y. Lafont and P. Taylor, *Proofs and types*, Camb. Tracts in Theoretical Computer Science 7, Cambridge University Press.

[Hagino 1987a] T. Hagino, A categorical programming language, Ph. D. thesis, University of Edinburgh.

[Hagino 1987b] T. Hagino, A typed lambda calculus with categorical type constructions. In D.H. Pitt, A. Poigné and D.E. Rydeheard, editors. *Category Theory and Computer Science, LNCS 283* pp 140-157.

[Hayashi 1985] S. Hayashi, Adjunction of semifunctors: categorical structures in nonextensional lambda calculus. *Theor. Comp. Sc. 41*, pp 95-104.

[Kleene 1936] S.C. Kleene, λ-definability and recursiveness. *Duke Math. J.* 2, pp 340-353.

[Lambek 1968] J. Lambek, A fixed point theorem for complete categories. *Mathematisches Zeitschrift 103* pp 151-161.

[Leivant 1989] D. Leivant, Contracting proofs to programs. In P. Odifreddi, editor. *Logic in Computer Science*, Academic Press, pp 279-327.

[Mendler 1987] N.P. Mendler, Inductive types and type constraints in second-order lambda calculus. *Proceedings of the Second Symposium of Logic in Computer Science.* Ithaca, N.Y., IEEE, pp 30-36.

[Mendler 1991] N.P. Mendler, Predicative type universes and primitive recursion. *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science.* Amsterdam, The Netherlands, IEEE, pp 173-184

[Newman 1942] M.H.A. Newman, On theories with a combinatorial definition of "equivalence". *Ann. of Math.* (2) 43, pp 223-243.

[Paulin 1992] Ch. Paulin-Mohring, private communication.

[Parigot 1988] M. Parigot, Programming with proofs: a second order type theory. *ESOP '88, LNCS 300*, pp 145-159.

[Parigot 1992] M. Parigot, Recursive programming with proofs. *Theor. Comp. Science* 94, pp 335-356.

[Reynolds and Plotkin 1990] J.C. Reynolds and G.D. Plotkin, On functors expressible in the polymorphic lambda calculus. In G. Huet, editor. *Logical Foundations of Functional Programming*, In 'The UT Year of Programming Series', Austin, Texas, pp 127-152.

[Wraith 1989] G.C. Wraith, A note on categorical datatypes In D.H. Pitt, A. Poigné and D.E. Rydeheard, editors. *Category Theory and Computer Science, LNCS 389* pp 118-127.

Girard's Paradox in lambda U

Leen Helmink, Eindhoven

Abstract

The presentation discusses a computer assisted construction of a proof of Girard's paradox in lambda U. Theoretical as well as practical issues will be addressed.

Computable interpretation of cross-cuts procedure

Hugo Herbelin (INRIA Rocquencourt and ENS Lyon)

Abstract

In the symmetric sequent calculus of Gentzen, several ways of eliminating cuts are possible. One of them, known as the cross-cut procedure, enjoys properties of symmetry. We will give an interpretation of this procedure in computable terms, in the paradigm of proofs as programs and cuts as communications between these programs. We will show from chosen examples that this sometimes gives more natural programs than the actually known interpretations of classical sequent calculus for arithmetic, such as LC or T. Coquand's interpretation in terms of games.

Typing in Pure Type Systems

Bert Jutting
Nijmegen

Abstract

In the theory of Pure Type Systems (PTS's) it is (to my knowledge) an open question whether Expansion Postponement holds. That is: Let $\vdash$ denote derivability in a PTS and $\vdash'$ derivability in the same PTS where the rule:

conv: $$\frac{c \vdash a : A, \qquad c \vdash B : s, \qquad A = B}{c \vdash a : B}$$

has been replaced by the rule

red: $$\frac{c \vdash' a : A, A > B}{c \vdash' a : B}$$

Is it true that $c \vdash a : A \Rightarrow c \vdash' a : B$ for some B with $A > B$?

Expansion Postponement is, among other things, important for proving correctness of typing algorithms.

In the talk I will propose an alternative definition giving rise to a different derivability relation $\vdash''$ satisfying:

$$c \vdash a : A \Leftrightarrow c \vdash'' a : A \text{ and } (A = s \text{ or } c \vdash'' A : s)$$

The importance of $\vdash''$ is stressed by the following properties:

1. For $\vdash''$ Expansion Postponement holds.
2. For $\vdash''$ an easy proof of strengthening is possible, which carries over to $\vdash$.
3. A straightforward typing algorithm for $\vdash$ can be derived from $\vdash''$.

Relating Logic Programming and Propositions-as-Types: A Logical Compilation

SUMMARY

James Lipton
Dept. of Mathematics
University of Pennsylvania

Aug. 15, 1992

Abstract

We analyze logic programs (Horn Clause programs and extensions) informally as *specifications* or *types* in the sense of the Curry-Howard isomorphism, rather than as programs. Using a realizability interpretation we develop a translation of these clauses to equational specifications. These give rise to various ways of associating non-deterministic terms or computations with this type that *satisfy* the logic program as a specification. By adapting a 1956 result of Nerode, (generalizing Herbrand-Gödel computability to term models) we are able to solve the equational specification directly over the term-model, and produce a (multivalued) Turing-machine index for the solution. We also define a realizability semantics in a partial applicative structure over the Herbrand Universe, and consider a non-deterministic or *disjunctive* λ*-calculus*. All interpretations are independent of any choice of logic programming interpreter. Rather, they transfer control and implementation features of proof search to control and implementation features of the recursion theorem. Completeness of the former is mapped to correctness of the latter.

Our translation provides a framework for integrating logic programming directly into a typed or untyped functional programming environment in a way that preserves the Curry-Howard content of typing. Our interpretations also provide new completeness theorems for various classes of logic programs. The results apply to Horn Logic Programs, and the Miller-Scedrov-Nadathur Uniform logic programming languages.

1 Introduction: Logic Programming as a specification Language

The results in this paper come from exploiting a largely unused uniformity that is present in most logic programming paradigms: the extraction of a witness to a query is uniform in the parameters and predicates in the program. By explicitly rewriting a logic program as a *realizability goal*, we search not for a specific witness, but a function that returns this witness for every choice of parameters. This reformulation is done first in the framework of an abstract form of the recursion theorem, adapted from [34], and then in terms of realizability over Feferman's partial applicative structures, where we can formalize logic programming languages, and where we are guaranteed the existence of the required partial recursive function via the appropriate fix-point theorem as in [1].

What we obtain is a compilation of logic programs into an equational specification of functional code via a generalized logic programming paradigm. The logic program is treated as a **type**, the computation process as one of finding a uniform inhabitant for the type. The entire development is implementation independent: we obtain not a particular evaluation strategy but a specification which gives different functional solutions according to different evaluation sequences. We also define a multivalued realizability which captures the nondeterminism of the declarative program directly. In this outline we develop one of these paradigms in detail, with an example, and give a quick sketch of the remaining ones.

1.1 Towards a Realizability Semantics for Logic Programs

In this section we develop an informal notion of realizability for logic programs. We begin with a logic program $\mathcal{P}$ decorated with *abstract realizers* α, which are taken as atomic evidence that the clauses are true (because the programmer says so). We first consider the first-order case: Horn or First Order Hereditarily Harrop logic programs, whose syntax is defined as follows. Let

$$\begin{array}{rcl} q & ::= & \top \mid a \mid q_1 \wedge q_2 \mid q_1 \vee q_2 \mid \exists x q \\ h & ::= & a \mid q \to a \mid h_1 \wedge h_2 \mid \forall x h \\ g & ::= & \top \mid a \mid g_1 \wedge g_2 \mid g_1 \vee g_2 \mid \exists x g \mid \forall x g \mid f \to g \\ f & ::= & a \mid g \to a \mid f_1 \wedge f_2 \mid \forall x f \end{array}$$

where a stands for an atomic formula. Then a **Horn clause program** is a finite set of closed h-formulas, and a **query** (or goal) for such a program is a q-formula (or a finite set of them). A **First Order Harrop program** is a finite set of closed f-formulas, and a **query** (or goal) for such a program is a g-formula (or a finite set thereof).

In the first part of this paper we will be considering an arbitrary Horn or FOHH program

$$\begin{array}{rcl} \alpha_1 : h_1(s_1) & \longleftarrow & T_1(t_1) \\ & \vdots & \\ \alpha_m : h_m(s_m) & \longleftarrow & T_m(t_m) \end{array} \tag{1}$$

where, s_i and t_i are -tuples of terms and where each clause is "decorated" with (abstract) *realizers* $\alpha_1, \cdots, \alpha_n$. These realizers supply atomic evidence that the corresponding clauses are being considered true.

Realizability over the Herbrand Universe Our first approach to extracting functional content from the program $\mathcal{P}$ defined above in (1) together with a query $Q[\vec{u}, \vec{v}]$ where the $\vec{u}$ are parameters and the $\vec{v}$ are variables, is to inhabit the Horn Clause specification directly with a realizer over the Herbrand Universe. This means finding a term e from a suitable partial combinatory structure $\mathbf{E}(\mathcal{H})$ defined over the Herbrand universe of the program satisfying

$$e : \forall \vec{u} \exists \vec{v}\ [\mathcal{P} \longrightarrow Q(\vec{u}, \vec{v}).] \tag{2}$$

As will be shown below, this entails finding a function $\hat{e}$ which, on a suitable domain D, satisfies the specification

$$(\forall \vec{u})\ (\mathcal{P} \longrightarrow Q[\vec{u}, \hat{e}(\vec{u})]).$$

We show how to produce a series of recursion equations specifying $\hat{e}$ from (2). Variants of Kleene's [19] and Nerode's [34] recursion theorem guarantee the existence of a solution in the set of partial recursive functions. We start with an example that shows how we obtain a specification for a multivalued function which captures the full non-determinism of the original declarative program seen independently of any choice of an interpreter.

1.1.1 An non-deterministic Example

We consider the realizability interpretation and the induced translations for the following Horn clause program, $\mathcal{P}$, equipped with "dummy realizers":

$$\alpha : \texttt{add(0,X,X)}. \tag{3}$$

$$\beta : \texttt{add(s(X),Y,s(Z))} \quad :- \quad \texttt{add(X,Y,Z)}. \tag{4}$$

Which means (see realizability definitions (2.5), below)

$$(\forall x)\ \alpha x : add(0, x, x) \tag{5}$$

$$(\forall x)(y)(z)(f) f : add(x, y, z) \to (\beta xyz) f : add(s(x), y, s(z)) \tag{6}$$

We trace through the compilation of this program to a specification of the non-deterministic function $f : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ satisfying

$$f(x)_0 + f(x)_1 = x$$

First, we translate this program to the following realizability goal, assigning what we will call below a $\langle 001\rangle$ template to the predicate add, that is to say, the character of an **input** to the third variable, and of an **output** to the first two:

$$\exists e\ (e : \forall \boldsymbol{u} \exists \boldsymbol{v} \exists \boldsymbol{w}\ \mathcal{P} \to add(\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{u}))$$

Unravelling this according to the definitions of realizability given below (with the notation $\underline{e}$ and $\widehat{e}$ for left and right projections of e),

$$(\forall \boldsymbol{u})[\underline{e}\boldsymbol{u} :\ \mathcal{P} \to add(\widehat{e}_0 u, \widehat{e}_1 \boldsymbol{u}, \boldsymbol{u})]$$

i.e.

$$(\forall \boldsymbol{u})(\forall \gamma)(\gamma :\ \mathcal{P} \to \underline{e}\boldsymbol{u}\gamma : add(\widehat{e}_0 \boldsymbol{u}, \widehat{e}_1 \boldsymbol{u}, \boldsymbol{u}))$$

Taking γ to be $\langle \alpha, \beta\rangle$ from (5) and (6) above, we have

$$\boxed{\underline{e}\boldsymbol{u}\langle \alpha, \beta\rangle : add(\widehat{e}_0 \boldsymbol{u}, \widehat{e}_1 \boldsymbol{u}, \boldsymbol{u})} \tag{7}$$

Now, unifying (7) on the third variable (that is to say, unifying on the template $\langle 001\rangle$) with the first clause of the original program (5), we have:

$$\alpha \boldsymbol{u} \quad : \quad add(0, \boldsymbol{u}, \boldsymbol{u}) \tag{8}$$

$$\underline{e}\boldsymbol{u}\langle \alpha, \beta\rangle \quad : \quad add(\widehat{e}_0 \boldsymbol{u}, \widehat{e}_1 \boldsymbol{u}, \boldsymbol{u}) \tag{9}$$

which has a solution $\widehat{e}$ if we choose:

$$\widehat{e}_0\boldsymbol{u} = 0 \qquad \text{and} \qquad \widehat{e}_1\boldsymbol{u} = \boldsymbol{u} \tag{10}$$

as well as $\underline{e}\boldsymbol{u}\langle\alpha,\beta\rangle = \alpha\boldsymbol{u}$. Now, assuming an e exists satisfying (7) we have, by applying the second clause (5) of the original program:

$$\forall\boldsymbol{u} : \beta[\widehat{e}_0\boldsymbol{u}][\widehat{e}_1\boldsymbol{u}][\boldsymbol{u}][\underline{e}\boldsymbol{u}\langle\alpha,\beta\rangle] \quad : \quad add(s(\widehat{e}_0\boldsymbol{u}), \widehat{e}_1\boldsymbol{u}, s(\boldsymbol{u})) \tag{11}$$

and applying (7) to $s(\boldsymbol{u})$

$$\forall\boldsymbol{u}\ \underline{e}s(\boldsymbol{u})\langle\alpha,\beta\rangle \quad : \quad add(\widehat{e}_0 s(\boldsymbol{u}), \widehat{e}_1 s(\boldsymbol{u}), s(\boldsymbol{u})) \tag{12}$$

where the choice of argument in (11) and (12) is dictated by unifying (7) and (6) on the last variable. Now (12) has a solution if $\widehat{e}$ satisfies the recursive equations

$$\widehat{e}_0(s(\boldsymbol{u})) = s(\widehat{e}_0\boldsymbol{u}) \qquad \text{and} \qquad \widehat{e}_1(s(\boldsymbol{u})) = \widehat{e}_1\boldsymbol{u}. \tag{13}$$

as well as the corresponding condition $\underline{e}(s\boldsymbol{u})\langle\alpha,\beta\rangle = \beta[\widehat{e}_0\boldsymbol{u}][\widehat{e}_1\boldsymbol{u}][\boldsymbol{u}][\underline{e}\boldsymbol{u}\langle\alpha,\beta\rangle]$ for $\underline{e}$. Leaving aside for the moment the requirements for the *evidence* $\underline{e}$, we have the following conditions for $\widehat{e}$:

$$\boxed{\begin{array}{rcl} \widehat{e}_0\boldsymbol{u} & = & 0 \\ \widehat{e}_1\boldsymbol{u} & = & \boldsymbol{u} \\ \widehat{e}_0(s(\boldsymbol{u})) & = & s(\widehat{e}_0\boldsymbol{u}) \\ \widehat{e}_1(s(\boldsymbol{u})) & = & \widehat{e}_1\boldsymbol{u} \end{array}} \tag{14}$$

We must be carreful about how we deal with the multivalued character of these equations. Taken at face value they logically imply the collapse of the underlying Herbrand Universe, since, for every $\boldsymbol{u}$, we have $0 = \widehat{e}_0\boldsymbol{u} = \boldsymbol{u}$.

For this reason, we interpret different conditions for $\widehat{e}_0$ and $\widehat{e}_1$ disjunctively. Before delving into our formalized treatment below, we give an informal treatment. Using $[\_\,\|\,\_]$ for (non-deterministic) disjunction, we can write this specification somewhat in the style of ML as:

$$\begin{array}{rcl} fun\ \ \widehat{e}\,0 & = & \langle 0,0\rangle \\ \mid\ \widehat{e}\,su & = & [\langle 0,su\rangle \,\|\, \langle s(\widehat{e}_0u),\widehat{e}_1u\rangle] \end{array} \tag{15, 16}$$

There is a natural execution model for this specification corresponding to every complete proof search strategy for the original program, for example a breadth-first traversal of the induced computation tree, until a leaf is found in normal form.

We also obtain a solution to this which captures the non-determinism of the specification itself by lifting the solution $\widehat{e}$ to a set-valued function $\tilde{e}$ (i.e. to the non-deterministic monad [31]) in the obvious way. Let *set* be the canonical lifting of functions $f : D \to D$ to maps $set(f) : \wp(D) \to \wp(D)$ between the power sets, given by:

$$set(f)(X) = \{f(x) : x \in X\}.$$

Then $\tilde{e} : D \to \wp(D)$ is given by

$$\begin{array}{rcll} \tilde{e}\,0 & = & \{\langle 0,0\rangle\} & (17) \\ \tilde{e}\,su & = & set([\lambda x.\langle 0,su\rangle \,\|\, \lambda x.\langle s(\mathbf{p}_0x),\mathbf{p}_1x\rangle])(\tilde{e}u) & (18) \\ & = & set(\lambda x.\langle 0,su\rangle)(\tilde{e}u)\ \cup\ set(\lambda x.\langle s(\mathbf{p}_0x),\mathbf{p}_1x\rangle)(\tilde{e}u) & \end{array}$$

where $\mathbf{p}_0$ and $\mathbf{p}_1$ are left and right components of the pairs in $\tilde{e}u$.

These approaches suggest several computational and realizability formalisations, each with its own fixpoint- or recursion theorem. We sketch one below adapted from Nerode's 1956 dissertation, which is in some sense independent of implementation (i.e. developed in terms of indices). Several concrete developments of the lambda calculus along these lines are sketched in the appendix.

1.2 Nerode-Kleene computability over term-algebras

The following development, adapted from Nerode's dissertation, provides an immediately applicable framework for generating and solving equations along the lines of the example studied above.

Definition 1.1 *A* **recursion calculus** *(a finite-signature one-sorted equational calculus) is a triple $e = (V, F, C)$ where V is a set of variables, and F and C are finite, nonempty sets of functions and constant symbols, respectively. We will also call (F, C) the e-***signature***.*

The **word e-algebra** W_e *is the Herbrand Universe of ground terms for the recursion calculus e.*
A finite set A of equations in the recursion calculus $e' = (V, F', C)$ is said to **overlay** *the word e-algebra W if $e = (V, F, C)$ and $F \subseteq F'$. We say such a set of equations* **distinguishes** *W if for any $w_1, w_2 \in W$,*

$$A \vdash w_1 \equiv w_2 \qquad \Rightarrow \qquad W \models w_1 \equiv w_2$$

We say A is **complete** *for W if for any function letter $f \in F'$ of arity k and any $w_1, \ldots, w_k \in W$ there is a $w_{k+1} \in W$ such that*

$$A \vdash f(w_1, \ldots, w_k) = w_{k+1}$$

We say A is a **partial definition** *over W if A overlays and distinguishes W. A is a* **definition** *if in addition it is complete.*

Definition 1.2 *If A is a (partial) definition over the word algebra W and f is a k-ary function symbol in A, we define the (partial) function*

$$[\![f]\!]_A : W^k \to W$$

by $[\![f]\!]_A(w_1, \ldots, w_k) = w_{k+1} \Leftrightarrow A \vdash f(w_1, \ldots, w_k) = w_{k+1}$.

It is easy to see that if A is a partial definition then $[\![f]\!]_A$ is well-defined, and is total if A is complete over W.

Definition 1.3 *The (partial) function $g : W^k \to W$ is* **definable** *if there is a (partial) definition A with a function symbol f of arity k such that $[\![f]\!]_A = g$.*

We now extend the notion of *recursive function* to a term algebra in a completely straightforward manner:

Definition 1.4 *Let $e = (F, C)$ be a finite signature, where F and C are ordered sets. Then the similarity type of e is a finite-support sequence of natural numbers τ with $\tau(n) = m$ iff there are m function symbols of -arity n. We define a Gödel numbering of the term algebra W_e (in fact* **the** *Gödel numbering induced by any encoding of sequences in $\mathbf{N}$ and by the ordering of F and C) to be a function $\# : W \to \mathbf{N}$ given by the standard ecoding $\#f(t_1, \cdots, t_n) = Seq(o(f), \#t_1, \cdots, \#t_n)$. A* **numbering** *of W_e is a bijection $I : \mathbf{N} \to W_e$ such that such that there are recursive bijections α and β with $\beta \circ \# = I^{-1}$ and $\alpha \circ I^{-1} = \#$.*

Definition 1.5 *Let I be a numbering of the word-algebra W. The (partial) function $g : W^k \to W$ is recursive over W if there is a recursive (partial) function $f : \mathbf{N}^k \to \mathbf{N}$ such that the following diagram commutes:*

$$\begin{tikzcd}
\mathbf{N}^k \arrow[r, "f"] \arrow[d, "I^k"'] & \mathbf{N} \arrow[d, "I"] \\
W^k \arrow[r, "g"'] & W
\end{tikzcd}$$

We then have the following theorem, proven in [34]:

Theorem 1.6 *The (partial) function $g : W^k \to W$ is definable iff it is recursive over W.*

The proof also shows that (an index for) the associated partial function $f : \mathbf{N} \to \mathbf{N}$ can be uniformly effectively computed from (codes for) the equations in a way similar to Kleene's original development of Herbrand-Gödel computability for partial recursive functions.

1.3 A nondeterministic term-model extension

Let e be a recursion calculus, and W_e its term model. Then the term model W_p of the calculus $e_p = (F \cup \{par\}, C \cup \{fail\})$ obtained by adding one distinguished binary function symbol *par* and the constant *fail* is called the *par-extension* of W_e. We call the set of equations $\mathcal{A}$ overlaying W_e a *par-extension theory* if the signature of $\mathcal{A}$ includes the function symbol *par* and the constant symbol $fail$, and if $\mathcal{A}$ includes the following equations.

1. (associativity of *par*)
$$par(par(x, y), z) = par(x, par(y, z))$$

2. (failure)
$$par(fail, x) = par(x, fail) = par(x, x)$$

3. (lifting) for each $f \in \hat{F}$ other than *par* of -arity $n > 0$
$$\begin{aligned} f(x_1, \ldots, x_{j-1}, par(y, z), x_{j+1}, \ldots, x_n) = \\ par(f(x_1, \ldots, x_{j-1}, y, x_{j+1}, \ldots, x_n), f(x_1, \ldots, x_{j-1}, z, x_{j+1}, \ldots, x_n)) \end{aligned}$$

Remark: A slight reformulation of the above will give us a syntactic variant of the *non-determinism triple* or monad construction (e.g. [31]). Let $\mathcal{C}_e$ be the category of (partial) e-algebras and algebra homomorphisms. Observe that if B is an e-algebra then so is the algebra B_a whose carrier $|B_a|$ is given by the set of terms

$$\{par(a_1, \cdots, a_m) : a_i \in |B| \cup \{fail\}\}$$

where $par(a_1, \cdots, a_m)$ is shorthand for $par(a_1, par(a_2, \cdots, par(a_{m-1}, a_m)))$. The algebra operations are induced by lifting (as given in the equations above), with constants a interpreted as $par(a, a)$. Let $T : \mathcal{C}_e \to \mathcal{C}_e$ be a functor defined by $T(B) = B_a$, with T's action on morphisms induced by the corresponding notion of lifting. Then let $\mu(B) : T^2(B) \to T(B)$ be given by

$$par(par(a_1, \cdots, a_m), b) \mapsto par(a_1, \cdots, a_m, b)$$

and $\eta(B) : B \to T(B)$ by $a \mapsto par(a, a)$ or $par(a, fail)$. Then (T, η, μ) is a monad.

The aim of this term-model construction is to reformulate equations of the sort produced in the example above (e.g. 14) which failed to distinguish the program signature, i.e. led to a collapse of the underlying Herbrand universe, as equations over a par-extension which constitute a legitimate partial definition over that extension. All that needs to be shown is that all the necessary definitions used in theorem 1.6 lift to the new structure.

Lemma 1.7 *Let I be a numbering of an e-term algebra W_e. Then there is a numbering I_p of its associated par-extension and a pair of recursive injections α and β from* $\mathbf{N}$ *to* $\mathbf{N}$ *such that for all a in W_e*

$$I^{-1}(a) = \alpha I_p^{-1}(a) \qquad \textit{and} \qquad I_p^{-1}(a) = \beta I^{-1}(a)$$

Since the par-extension of W_e is just another term-algebra, the main theorem 1.6 states that a function is definable over the par-extension of W_e iff it is partial recursive.

1.3.1 Multivalued Turing Machines

In the next section we define a procedure which when applied to a logic program and uniform query $\mathcal{P} \to Q$ produces a set of equations $\mathcal{A}$ which give partial definition over the par-extension of the Herbrand universe of $\mathcal{P}$. By the main theorem of this section, every $\mathcal{A}$-defined function f has an associated Turing-machine index e_f in the sense of definition (1.5). We now define the associated multivalued Turing machine R_f in the obvious way:

$$R_f(n) = par\_set(e_f(n))$$

where

$$par\_set(n) = \{I^{-1}(a) : a \,\varepsilon\, I_p(n)\}$$

and where the relation ε is given by

$$x \,\varepsilon\, par(y, z) \Leftrightarrow x = y \vee x \varepsilon z.$$

Our development is not particularly different than defining r.e. set valued recursive functions by associating, with any partial recursive $f : \mathbf{N} \to \mathbf{N}$, the function $F : \mathbf{N} \to \Re$ given by $F(x) = S_e$ where S_e is the domain of the e-th Turing machine. The only significant difference is that the numbering of sets is more natural since it is induced by (Gödel numbering of) terms used to defined sets of values.

generic variables

We must also deal with fresh constants or variables introduced by the **generic** step for SLDR-computation (see below) or by equations induced by *unbalanced* programs: those with occurrences of a free variable in the head of a clause that does not also occur in the tail. Thus, we may need to generate equations of the form e.g.

$$ea = y \tag{19}$$

where a is a constant, or a term in which y is not free. In order to solve these over par-extensions we must include special generic-variable terms $gen(c_i)$ $(i > 0)$ where the c_i are fresh constants. Then equations like (19) are rewritten

$$ea = gen(c_i)$$

where c_i is the first such constant not already used. When we lift the partial recursive functions computed via theorem (1.6) to nondeterministic functions, we interpret $gen(c_i)$ as the set of all terms over the original Herbrand model. These generic variables appear as uppercase logic variables in the lambda calculi introduced in the appendix.

1.3.2 Term-models with choice operators

A useful variant of the term-model construction just given, is the introduction of ternary function symbols

$$cho(\sigma, a, b)$$

in lieu of $par(a, b)$, with σ an "unspecified" choice index $\in \{0, 1\}$. *cho* satisfies the additional equations

$$cho(0, a, b) = a \qquad cho(1, a, b) = b$$

This indexing device can also be added to the disjunctive lambda calculus (see appendix) to preserve unrestricted β-reduction and the Church-Rosser property. For convenience, we can extend this in the obvious way to terms

$$cho(\sigma, a_1, \cdots, a_n)$$

where σ is in $\{0, 1, \cdots, n-1\}$ and e.g. $cho(1, a, cho(1, b, c)) = cho(2, a, b, c)$. We will call an assignment of specific values to all σ in a choice term-model an *instantiation* of the model, and by abuse of language, we will also call them instantiations of the associated par-terms in the par-extension. With such instantiations, the multivalued functions computed above now become single valued functions over the Herbrand Universe. These are also called *instances* of the corresponding par- or multi-valued functions.

2 SLDR-computation of Equations

We now describe the SLDR-computation algorithm for generating equations ovber the par-extension of the Herbrand Universe (or a generic extension).

Definition 2.1 ($\mathcal{F}$ - unification) *Let Σ be a signature and $\mathcal{F}$ a set of function symbols disjoint from those in Σ. Let s and t be terms over the signature $\Sigma \cup \mathcal{F}$. A term s over this signature* *is called a Σ-term (or just a* pure term*) if no symbol in $\mathcal{F}$ occurs in s, otherwise it is called an* $\mathcal{F}$ -term. *An $\mathcal{F}$* -term *s is said to be* **rigid**, *or in* head form *if s is $e(t_1, \cdots .t_n)$ for some $e \in \mathcal{F}$.*

The set $\mathcal{C}(s,t)$ of $\mathcal{F}$ -**unification constraints** *for $\langle s,t\rangle$ is a finite set set of equations defined according to the structure of s and t as follows:*

Case 1. s and t pure: then $\mathcal{C}(s,t)$ is the equational representation of the most general unifier of s and t: namely a finite set of **basic** *equations, i.e.a set of equations of the form $\{\cdots x_i = \tau_i \cdots\}$ where the x_i are variables not occurring in any of the terms τ_j, representing the substitution $x_i \mapsto \tau_i$.*

Case 2. s or t $\mathcal{F}$ -terms: If either s or t is **rigid**, *$\mathcal{C}(s,t) \equiv \{s = t\}$. Otherwise we consider the usual cases found in first-order unification:*

- *$s = f(s_1, \cdots, s_n)$ and $t = a$ or $t = g(t_1, \cdots, t_m)$ where a, f, g are in Σ and f is distinct from g: then $\mathcal{C}(s,t) = \{s = fail, t = fail\}$.*
- *$s = f(s_1, \cdots, s_n)$ and $t = f(t_1, \cdots, t_n)$ where $f \in \Sigma$: then $\mathcal{C}(s,t) = \bigcup \mathcal{C}(s_i, t_i)$ (with the proviso that if fail occurs in $\mathcal{C}(s,t)$ then this is equivalent to $\mathcal{C}(s,t) = \{fail\}$).*

any basic equations in $\mathcal{C}(s,t)$ contributed as a result of case (1) will be called **pure** *unification constraints.*

Scheduling We do not go into the details of scheduling here. Since the compilation procedure below always terminates, bad scheduling cannot cause divergence. Let us remark, however that the development is in the style of constraint logic programming, so that failure of unification may not be noticed unless the generated equations are periodically analyzed for consistency with respect to the (decidable) first order theory of equality over the Herbrand Universe. (Identities involving introduced function symbols never introduce inconsistencies). For example, an attempt to unify $Q(f(x))$ and $Q(y)$ in the presence of the constraint $\{y = g(a)\}$ may result in success followed by the expansion of the constraint set to $\{y = g(a), y = f(x)\}$ which must eventually generate failure. In prolog, this happens at unification time, and there is no reason why this could not be done with SLDR-resolution. Then we must modify the definitions above appropriately to include $\mathcal{F}$ - $\mathcal{C}$ - **unification** of s and t, i.e. unification in the presence of constraints $\mathcal{C}$. We just apply $\mathcal{C}$ to s and t prior to carrying out $\mathcal{F}$ -unification.

We now describe the SLDR-computation process. First a few conventions regarding introduction of function symbols and scheduling.

Introduction of function symbols: We make the following assumptions on the predicates, to facilitate their functional interpretation: all unary predicates $P(a)$ are treated as predicates in two variables,of 2 variables $P(a, true)$ so that the associated function e_P satisfies $e_P(a) = true$ when $P(a)$. 0-ary predicates are just treated as Boolean constants. n-ary predicates $Q(t_1, \cdots, t_n)$ are assigned an input-output *template*: a binary string of length n defining some slots as inputs, and the remaining ones as outputs, i.e., as components of the associated function value. Because of the inherently relational nature of the induced multivalued equations, there is no need to consider more than one template per predicate and (with the exception of the top-level query) no reason to pick one over another. The equations generated by the SLDR-procedure below will work for any choice. For efficiency in solving these equations and ease of representation, it may prove useful to introduce more than one function for a given predicate corresponding to different templates, (e.g. to associate subtraction with $add(in_1, out, in_2)$ and addition with $add(in_1, in_2, out)$), but this can be done at the time of solving the equations, and *it is not required for the theoretical development or the results below.*

In what follows, we will describe the local state of a program $\alpha : \mathcal{P}$ with distinguished clause C, current goals $\tau_1 : Q_1, \ldots, \tau_n : Q_n$, equations $\mathcal{C}$ as a node on the SLDR tree with the notation

$$\langle \mathcal{C}, \alpha\colon \mathcal{P}, C \vdash Q_1, \ldots, Q_n \rangle$$

and we show how to develop the subsequent nodes of the tree by induction on the structure of the *current query* Q_1.

Definition 2.2 *Let $\mathcal{P}$ be a first-order program with assigned realizer $\hat{\alpha}$, and suppose one of the clauses of $\mathcal{P}$ is*

$$\beta : T(t) \to Q[r, s]$$

where $[r, s]$ is any breakdown and rearrangement of the predicate $Q(t_1, \cdots, t_n)$ (e.g. $Q(a, b, c)$ with $x = b$ and $y = (a, c)$). Further suppose that the current query is $Q[x, y]$ i.e. the current state is

$$\langle \mathcal{C}, \alpha\colon \mathcal{P}, \beta : T(t) \to Q(\boldsymbol{v}, s) \vdash Q[x, y] \rangle$$

and that the predicate letter Q has not already occurred as a goal. Then, we introduce a function symbol e (or e_Q), rewrite the current state of the program as

$$\langle \mathcal{C}, \alpha\colon \mathcal{P}, \beta : T(t) \to Q(\boldsymbol{v}, s) \vdash \underline{e}\hat{\alpha} : Q[u, \hat{e}\boldsymbol{u}] \rangle$$

Then the following is the SLDR-reduction step **backchain***:*

$$\langle \mathcal{C}, \alpha : \mathcal{P}, \beta : T(t) \to Q[\boldsymbol{v}, s] \ \vdash \ \underline{e}\alpha' : Q[u, \hat{e}\boldsymbol{u}] \rangle$$

$$\downarrow$$

$$\langle \mathcal{C} \cup \{\mu, \ \hat{e}(\boldsymbol{u}) = s, \ \underline{e}\alpha' = \beta\}, \alpha : \mathcal{P} \vdash \gamma : T(t) \rangle$$

where μ is the set of pure unification constraints (as defined above in 2.1) generated by $\mathcal{F}$-unification of $Q[\boldsymbol{v}, s]$ and $Q[u, \hat{e}\boldsymbol{u}]$.

For the sake of thoroughness we have shown how to process the evidence (e.g. $\underline{e}$ above) along with the witnesses. As remarked above, for self-realizing classes of formulas there is no point to carrying along this extra information, so we omit these terms in the remaining proof steps[1]

We assume that the program is written in such a way that there is never more than one clause with a given predicate occurring in the head. Any logic program can be so rewritten by the use of disjunction in the tail of a clause, and the possible addition of equational goals. For example

$$(Q(s) \longleftarrow S) \wedge (Q(t) \longleftarrow T)$$

can be rewritten to

$$Q(x) \longleftarrow ([x = s, S] \vee [x = t, T]).$$

for a fresh (vector) variable x.

[1]their sole purpose here, other than to give another motivation for considering hereditarilky self-realizing classes of formulas a logic programming languages, is to make possible the treatment of more general classes via the generation of realizability constraints. If the program is **not** self-realizing, it is only true in the induced realizability model. But this will be taken up elsewhere!

Definition 2.3 *Let $\mathcal{P}$ be as above and suppose a goal is of the form $S(s) \vee T(t)$. The following is the SLDR-reduction step* **co-satisfy**

$$\begin{array}{ccc} & \langle \mathcal{C}\,,\,\mathcal{P}\,\vdash S \vee T\rangle & \\ & \swarrow \quad \searrow & \\ \langle \mathcal{C}\,,\,\mathcal{P}\,\vdash S\rangle & & \langle \mathcal{C}\,,\,\mathcal{P}\,\vdash T\rangle \end{array}$$

The steps corresponding to all the remaining ones in e.g. FOHH logic programming, e.g. **augment** and **generic** in [24] remain unchanged. **augment** is

$$\begin{array}{c} \langle \mathcal{C}\,,\,\mathcal{P}\,\vdash A \to B\rangle \\ \downarrow \\ \langle \mathcal{C}\,,\,\mathcal{P}\,, A \vdash B\rangle \end{array}$$

and **generic**

$$\begin{array}{c} \langle \mathcal{C}\,,\,\mathcal{P}\,\vdash \forall x A\rangle \\ \downarrow \\ \langle \mathcal{C}\,,\,\mathcal{P}\,\vdash A(c)\rangle \end{array}$$

where c is a fresh constant. The most important step in the SLDR derivation of equations is the **recursion** step, which occurs every time a goal, e.g. $Q(t_1, \cdots, t_{n+m})$ occurs for a second time on the same path of an SLDR-tree. Then a witnessing function e_Q has already been introduced for it, with the condition

$$\forall Q(x_1, \cdots, x_n, (e_Q(x_1, \cdots, x_n))_1, \cdots, (e_Q(x_1, \cdots, x_n))_m)$$

so the goal is *immediately satisfied*, with the corresponding $\mathcal{F}$-unification equations added to $\mathcal{C}$:

recursion

$$\begin{array}{c} \langle \mathcal{C}\,,\,\mathcal{P}\,, \vdash Q(t_1, \cdots, t_{n+m})\rangle \\ \downarrow \\ \langle \mathcal{C} \cup (\bigcup_{i=1}^{n} \mathcal{C}\,(x_i, t_i)) \cup (\bigcup_{j=1}^{m} \mathcal{C}\,((e_Q(x_1, \cdots, x_n))_j, t_j))\, \mathcal{P}\,, \vdash \ \rangle \end{array}$$

The search terminates when every goal on the right of the turnstile has a set of recursion equations specifying the corresponding realizer in $\mathcal{C}$. Termination depends only on the predicate *letters* occurring in (a first-order) program. Call a predicate letter A *active* if it has not yet recurred in the sense of the recursion step above. Then, a give predicate B can only occur on the right of a turnstile if:

- it has never occurred before,

- it recurs on some path for the first time

- it is recurring again on some path because it is a subgoal associated with an *active* goal A (e.g. $\longleftarrow B, C_1, \cdots, C_n \vdash A$), for if A were recurrent we would satisfy it using *recursion* rather than *backchain*.

Thus the number of recurrences of an inactive letter is bounded by 1+ the total number of predicate letters in the program. We omit the straightforward (but tedious) details in this summary.

In particular, termination is guaranteed irrespective of termination of SLD resolution, even if the original program never halts for any input!, in which case a solution will be a code or term denoting the totally divergent function, e.g. a least-fix-point solution to the equation $e = e$.

We also leave for a fuller development of this paper a description of the compilation of the multivalued equations for all introduced function variables. Essentially, the equations on different paths through a node of an SLDR-tree are processed separately and then merged via disjunctive expressions

$$\hat{e}x = [t_1 \| \cdots \| t_n]$$

or using the formal terms in a par-extension of the Herbrand Universe. The particular par-terms that arise are clearly determined by the order in which the equations on different branches are processed.

2.1 Semantics and correctness

Formalizing Realizability and computation over a Herbrand Universe

We will let $\mathcal{L} = \mathcal{L}(\mathcal{P})$ denote the language of the logic program. Then define $\mathcal{H}_\mathcal{P}$ to be the Herbrand Universe of the program (the set of ground terms over $\mathcal{L}$). We now build a partial applicative theory around $\mathcal{H}_\mathcal{P}$ or its associated par-extension along lines similar to e.g. ([1], [40]). We will take Beeson's formulation of Feferman's theory **APP** (which Beeson calls **PCA+**), with a primitive unary postfix predicate $\downarrow$ for "terms that denote", together with the standard *partial* combinators **s** and **k**, pairing and unpairing, a unary integer sort **N** and a 4-ary definition by integer cases [2] operator **d**. The key results we will need here about APP are that it satisfies combinatory completeness (admits lambda abstraction), and Kleene's (single and double) **recursion theorems**, admits "strong" definition by cases, and satisfies the numerical and term existence property. See ([1],or [40]) for details.

Definition 2.4 *Let $\mathcal{P}$ be a (Horn or FOHH) logic program decorated with abstract realizers $\hat{\alpha} = \langle \alpha_1, \ldots, \alpha \rangle$. Then we define the associated applicative theory $\mathbf{E}(\mathcal{H}_\mathcal{P})$ extending* **APP** *to include the following:*

- *constants c, f for every constant* `c` *and n-ary function symbol* `f` *in $\mathcal{L}$. We call these "symbols imported from $\mathcal{L}$", and also denote them $(c)^*$ and $(f)^*$. We also include the abstract realizers α_i directly as constants.*

- *constants t for every ground term* `t` *from $\mathcal{L}$. This means that e.g. for constants a, b, c and function symbols f, g imported from $\mathcal{L}$ the terms* `f(a,g(b,c))` *and*

$$((((fa)g)b)c) \stackrel{\text{def}}{\equiv} App(App(App(App(f, a), g), b), c)$$

[2] $\mathbf{d}xyzw \ \equiv \ if\ x = y\ then\ z\ else\ w$

are syntactically distinct objects in $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ *, (although they are identified via new axioms below).*

- *A unary predicate* H *denoting the* universe of the interpretation.
- *for every term* t *and function symbol* f *imported into* **APP** *from* $\mathcal{H}_{\mathcal{P}}$ *, and for each abstract realizer* α_i*, the axioms*

$$t\downarrow, \qquad H(t), \qquad f\downarrow \qquad \alpha_i\downarrow \qquad (1 \le i \le n)$$

and for each n-ary function symbol f *and terms* $t_1, \ldots, t_n$ *imported from* $\mathcal{L}$ *,*

$$(ft_1)\downarrow, \qquad (ft_1)t_2,\downarrow \qquad \cdots \qquad ft_1 \cdots t_n = (f(t_1, \ldots, t_n))^*$$

- *The constant* fail *together with the axiom* fail$\downarrow$.
- *All schemas in* **APP** *are to be extended to the new terms (e.g.* $t\downarrow \wedge \forall x\, A(x) \rightarrow A(t)$ *for each term* t *).*

We extend the $*$*-embedding of terms to formulas in the obvious way:* $p(t_1, \ldots, t_n)^* = p((t_1)^*, \ldots, (t_n)^*)$, $(\varphi * \theta)^* = (\varphi)^* * (\theta)^*$ *for* $*\in\{\rightarrow, \wedge, \vee\}$*, etc. We define* $\in$*to be* $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ *together with the embedded program itself (i.e. in the form* $\{(theta)^* : \theta\in\mathcal{P}\}$*) as a set of nonlogical axioms.*

We now develop realizability style semantics over the applicative theories just defined in a manner analogous to e.g. [9, 1, 40], but restricting ourselves to interpreted formulas from the language of the original program. Although we have carried out this interpretation in order to have one theory to discuss both program and evidence, one should perhaps think of this also as a realizability with a distinct *object language*, namely that of the program, and a *realizing metalanguage*, namely $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ or $\mathbf{E}(\mathcal{P})$.

Definition 2.5 (Syntactic Realizability over $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$) *For a term* t *of* $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ *and a formula* θ *over the language of the original program* $\mathcal{L}$*, we define the binary relation* $t : \theta$*, (*t **realizes** θ*) by cases and by induction on the structure of* θ *as follows:*

$$\begin{array}{rcl}
\alpha_i : \theta & & \text{if } \theta \text{ is the } i\text{-th clause of the program} \\
t : \theta \wedge \varphi & \stackrel{\text{def}}{\equiv} & \mathbf{p}_0 t : \theta \wedge \mathbf{p}_1 : \varphi \\
t : \theta \vee \varphi & \stackrel{\text{def}}{\equiv} & \mathbf{N}(\mathbf{p}_0 t) \wedge (\mathbf{p}_0 t = 0 \rightarrow \mathbf{p}_1 t : \theta) \wedge (\mathbf{p}_0 t \neq 0 \rightarrow \mathbf{p}_1 t : \varphi) \\
t : \theta \rightarrow \varphi & \stackrel{\text{def}}{\equiv} & (\forall \xi)[\xi : \theta \rightarrow t\xi : \varphi] \\
t : (\exists_{x\in D})\theta(x) & \stackrel{\text{def}}{\equiv} & D(\mathbf{p}_0 t) \wedge \mathbf{p}_1 t : \theta[x/\mathbf{p}_0 t] \\
t : (\forall_{x\in D})\theta(x) & \stackrel{\text{def}}{\equiv} & \forall u\, D(u) \rightarrow tu\downarrow \wedge tu : \theta([x/u])
\end{array}$$

We now can state our main representation theorem for first-order logic programs. We use the following notational convention. If t is a term in $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$, we write $t = \langle t_1, \ldots, t_n \rangle$ to denote iterated pairing with association to the right. Thus $\mathbf{p}_0 t = t_1$, $\mathbf{p}_0(\mathbf{p}_1 t) = t_2$, etc.

Theorem 2.6 *Let* $\mathcal{P}$ *be a first-order program, with generic query* $\theta[u, v]$*. Then*

1. *Any instance e of a SLDR-computed term e in* $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ *provably realizes the program in the sense that for some D definable in* $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ *:*

$$\mathbf{E}(\mathcal{H}_{\mathcal{P}}) \vdash e : (\forall \vec{u} \in D)(\exists \vec{v} \in D)(\mathcal{P} \to \theta[\vec{u};\vec{v}])$$

2. *In particular, there is a term* $t = \langle t_1, \ldots, t_n \rangle$ *in* $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$ *which is (the *-image of) an instance of an SLDR computed term and which maximally satisfies the program as a specification in the sense that for any input u in* $\mathcal{H}$ *for which*

$$\mathcal{P} \vdash \exists \vec{v} \theta[\vec{u};\vec{v}]$$

we have

$$\mathbf{E}(\mathcal{H}_{\mathcal{P}}) \vdash (\mathcal{P} \to \theta[\vec{u}; t\vec{u}])$$

Such a witnessing term t is faithful to the program in that

- *whenever* $(t_i u_1 \cdots u_m)\downarrow$ *the values* $(t_i u_1 \cdots u_m)$ *are (modulo the *-translation between language and metalanguage) correct answers for the original program* $\mathcal{H}$ *.*
- *whenever values* $v_1, \ldots, v_n$ *exist for a set of inputs* $u_1, \ldots, u_m$*, we have* $(t_i u_1 \cdots u_m)\downarrow$*.*

The theorem follows from the fact that *all logic programming languages considered here are self-realizing classes of formulas*(see e.g. [1, 40]). Thus whenever

$$(\mathcal{P} \to \theta[u_1, \ldots, u_m; (t_1 u_1 \cdots t_1 u_m), \ldots, (t_n u_1 \cdots t_n u_m).])$$

is realized provably in $\mathbf{E}(\mathcal{H}_{\mathcal{P}})$, it is provable, and conversely. It is shown in e.g. [1] that all the key properties of **APP** are preserved in extensions by self-realizing theories, including the soundness of most realizability notions (in particular, ours), the existence property, and the recursion theorem.

It is clear that the structure of SLDR-computation is geared towards preserving the computational content of the logic programs without selecting an arbitrary evaluation strategy. To make this precise we need to compare proof search strategies (selection and branching rules) with evaluation strategies of the resulting terms. The notational machinery for this is cumbersome and is omitted here, although the results and ideas are straightforward.

Corollary 2.7 *Suppose terms e and t from the theorem above are instances of Nerode-Kleene solutions to a finite set of equational constraints* $\mathcal{C}$ *over the disjunctive term calculus obtained by SLDR-computation. Then these terms preserve the computational content of the logic program in the sense that every fair evaluation instantiation strategy for the resulting terms in a choice-extension of the Herbrand model corresponds to a complete proof search strategy for the logic program.*

3 Other realizability interpretations

3.1 Adding a Domain or Type variable: Constraint Logic programming

We provide a brief sketch of the way a modified realizability enables us to capture constraint logic programming and at the same time ensure a *total* realizability witness, by constraining the *domain* of the computed realizer. The idea is to formally add an *existentially* quantified domain variable D when formulating the realizability goal for $\mathcal{P}$:

$$e : \exists D (\forall \boldsymbol{u} \in D)(\exists v \in D)(\mathcal{P} \rightarrow Q(u,v)).$$

This formulation does not require a partial applicative structure, and can be developed in total type theoretic frameworks such as, e.g. Martin-Löf type theory [5, 40, 1, 28].
It can also be developed using Kreisel-Troelstra realizability over HAS or IZF (see e.g. [25]). In this case constraints are added to the list of goals to be solved, and determine the instantiation of the variable D.

3.2 A Kripke model for first-order logic programs

We can capture the notion of partial realizability of logic programs formalized directly over the Herbrand model using Nerode-Kleene computability by considering pairs (e, D) where D is a subset of the Herbrand Universe on which e converges as follows.
We define a Kripke model $\mathcal{K}$ with the following carrier set:

$$K = \bigcup K_{\vec{x},\vec{y}} \text{ where } K_{\vec{x},\vec{y}} = \{(e, D, \vec{x}, \vec{y}) : \forall \vec{u} \in D\; e\vec{u} \downarrow\}$$

where $\vec{x}, \vec{y}$ range over all tuples of variables $x_1, \ldots, x_k$ and $x_{k+1}, \ldots, x_{k+m}$ $(n, m > 0)$ and with order given by $(e, D, \vec{x}, \vec{y}) \leq (e', D', \vec{x}, \vec{y})$ if e extends e' (i.e. $\forall x e'(x) \downarrow \rightarrow ex \downarrow \wedge e'(x) = ex$) and D is a subset of D. Nodes $(e, D, \vec{x}, \vec{y})$ may only force atomic formulas with free variables matching the tuples $\vec{x}, \vec{y}$. Atomic forcing is given by

$$(e, D, \vec{x}, \vec{y}) \Vdash A[\vec{x}, \vec{y}] \qquad \Leftrightarrow \qquad \forall \vec{u} \in D\; \mathcal{P} \vdash A[\vec{u}, e\vec{u}]$$

where $e\vec{u}$ is a tuple of length equal to $\vec{y}$. We introduce the following notion of cover: The set $S \subseteq K$ is a cover of $(e, D, \vec{x}, \vec{y})$ if for every $(e', D', \vec{x}, \vec{y})$ in S, (e', D') is above S and $D \subseteq \bigcup_{D' \in S}$. Then we define forcing of disjunctions via covers (see e.g. [40]) and obtain a model in which for all first-order goals A

$$(e, D, \vec{x}, \vec{y}) \Vdash A[\vec{x}, \vec{y}] \qquad \Leftrightarrow \qquad \forall \vec{u} \in D\; \mathcal{P} \vdash A[\vec{u}, e\vec{u}]$$

Using this semantics we obtain another completeness theorem for ther SLDR translation (provide(1 27 16 1) d $\mathcal{P}$ is assumed to be Horn or Hereditarily Harrop): *every node $(e, D, \vec{x}, \vec{y})$ in $\mathcal{K}$ forcing a query $Q[\vec{x}\vec{y}]$ is extended by some instance of an SLDR-computed term e′, and every SLDR-computed term occurs in a node of $\mathcal{K}$.*

3.3 Realizability by multivalued functions

As we illustrated by example above, we can also choose constructive set theory (IZF) as our metatheory, with an applicative structure embedded in it, as in e.g. McCarty's ([25]). We are then able to construct the realizability interpretation of IZF as a basic model, (also known as the realizability universe, or –with some variation–the effective topos, see e.g. [16]). Our realizers live in this model, but will not be the terms of the formalized applicative structure. They are *sets* and multivalued functions definable in IZF, with the following possible realizability definitions induced by term realizability : over the logic program, which we call strong and weak realizability ($\underset{\sim}{\mathrm{s}}$ and $\underset{\approx}{\mathrm{w}}$). Strong realizability by sets of realizers is just

$$e \underset{\sim}{\mathrm{s}} \theta \overset{\text{def}}{\equiv} (\forall x \in e)(x : \theta)$$

Weak realizability is as follows. Conjunction and implication and existence are as before. The interesting clause is disjunction.

$$e \underset{\sim}{\mathrm{w}} \theta \vee \varphi \overset{\text{def}}{\equiv} (\forall x \in e)[(x : \theta) \vee (x : \varphi)]$$

and

$$e : \exists x \theta(x) \overset{\text{def}}{\equiv} (\forall \xi \in e)(\xi_0 : \theta(\xi_1)).$$

Towards a non-deterministic realizability of logic programs

If we use the more syntactic formulations of multivalued functions described above and in the appendix, then order of the terms, and the possible presence of a *fail* token suggest more complex definitions, to capture negation as failure

$$[t_1 \| t_2] : \theta \vee \varphi \overset{\text{def}}{\equiv} t_1 : \theta_\perp \wedge t_2 : \varphi_\perp \wedge (t_1 = \mathit{fail} \rightarrow t_2 : \varphi) \wedge (t_2 = \mathit{fail} \rightarrow t_1 : \varphi)$$

where

$$t : (\theta)_\perp \overset{\text{def}}{\equiv} t : \theta \vee t = \mathit{fail}$$

and where for any predicate $Q(x_1, \cdots, x_n)$, $Q[x_i/\mathit{fail}]$ is true. We are able to obtain an analogue of theorem 2.6 for multivalued realizability, discussed in the [21].

4 Conclusion

Using realizability interpretations of logic programs we can translate them into disjunctive terms that (weakly) inhabit these programs as types, or specifications in the Curry-Howard sense. These translations maintain the declarative meaning of the program while leaving control features untouched. These are tranferred to the evaluation-control problems: the way in which the recursion theorem and disjunctive branching conditions are evaluated in the target model. This points to the possibility of studying control features of logic programming in the context of functional programming with nonlocal control operators. It also gives a natural way of associating domain-theoretic interpretations to logic programs. Our translations also transform logic programs into first and higher order constraints over various kinds of applicative structures. This also permits the formalization within logic programming itself of constraint solving in a new way. Many new questions need to be addressed here: which constraints generated by logic programs in the style discussed here have solutions over different typed calculi. Can we identify a (typed) logic programming language corresponding to various subrecursive classes? How can we modify our realizability calculus to capture non-local control more efficiently and clearly? Perhaps most importantly of all, our work suggests that a major task ahead of us is to understand equation and constraint-solving (together with feasibility questions) over applicative structures along the lines initiated by Statman and Tronci ([38, 39]). This work must combine type inference and simultaneous constraint solving. This issue is discussed in more depth in [21].

In particular, higher order programming is likely to require the simultaneous solution of domain equations and constraint sets over continuous or computable functions on the indicated domain.

Appendix

The disjunctive calculus

Following the example computed earlier, we can extend the term structure of the lambda calculus or of a partial combinatory calculus such as **APP** to include disjunctive terms $[t_1 \| t_2]$. We also consider the addition of generic or logic variables X (which can also be thought of as labelled contexts) standing for all possible (non-disjunctive) instances. We consider two developments here: **pure disjunctive calculus**

$$t \quad ::= \quad x \mid X \mid (t_1 t_2) \mid \lambda x.t \mid [t_1 \| t_2]$$

indexed disjunctive calculus

$$t \quad ::= \quad x \mid (X)^\sigma \mid (t_1 t_2) \mid \lambda x.t \mid [t_1 \underset{\sigma}{\|} t_2]$$

The second calculus allows us to reason about disjunctive lambda terms with a "generic" evaluation index σ which is assumed to select one of the disjuncts, which admits full-blown β reduction. In the pure calculus we must lift disjunctions to the top (using rule 22 and rule 30 prior to beta reducing with rule 23. We now give reduction rules for both calculi. The optional rules are included to better handle the simplification of functions computed from logic programs, but do not seem eessential for the lambda calculi themselves.[3]

pure disjunctive calculus

$$\begin{array}{rcll}
[s \| t]\, u & \rightsquigarrow & [su \| tu] & \quad (20)\\
(\lambda x.s)u & \rightsquigarrow & s[x/u] \qquad u \text{ non-disjunctive} & \quad (21)\\
\lambda x.\,[s \| t] & \rightsquigarrow & [\lambda x.s \| \lambda x.t] & \quad (22)\\
(\lambda x.s)\,[u \| v] & \rightsquigarrow & [(\lambda x.s)u \| (\lambda x.s)v] & \quad (23)\\
\langle\, [u \| v]\, , z\rangle & \rightsquigarrow & [\langle u, z\rangle \| \langle v, z\rangle] & \quad (24)\\
\langle z,\, [u \| v]\, \rangle & \rightsquigarrow & [\langle z, u\rangle \| \langle z, v\rangle] & \quad (25)
\end{array}$$

indexed disjunctive calculus

$$\begin{array}{rcll}
[s \underset{\sigma}{\|} t]\, u & \rightsquigarrow & [su \underset{\sigma}{\|} tu] & \quad (26)\\
(\lambda x.s)u & \rightsquigarrow & s[x/u] & \quad (27)\\
\langle\, [u \underset{\sigma}{\|} v]\, , z\rangle & \rightsquigarrow & [\langle u, z\rangle \underset{\sigma}{\|} \langle v, z\rangle] & \quad (28)\\
\langle z,\, [u \underset{\sigma}{\|} v]\, \rangle & \rightsquigarrow & [\langle z, u\rangle \underset{\sigma}{\|} \langle z, v\rangle] & \quad (29)
\end{array}$$

[3]The author has recently learned of substantial work in disjunctive calculi by Piperno, Liguori, and Dezani, among others, which may provide a better framework for nondeterministic term extraction from logic programs.

optional rules

$$f\,[u \,\|\, v] \quad\rightsquigarrow\quad [fu \,\|\, fv] \tag{30}$$

$$[u \,\|\, fail] \quad\rightsquigarrow\quad u \tag{31}$$

$$[fail \,\|\, u] \quad\rightsquigarrow\quad u \tag{32}$$

$$[u \,\|\, u] \quad\rightsquigarrow\quad u \tag{33}$$

References

[1] Beeson, M. J. [1985a], *Foundations of Constructive Mathematics*, Springer-Verlag, Berlin.

[2] Beeson, M. J. [1985b], "Proving programs and programming proofs", in *Logic, Methodology, and Philosophy of Science* VII, North-Holland, Amsterdam.

[3] Cerrito, Serenella, [1992], "A linear axiomatization of negation as failure", in the *Journal of Logic Programming*, 12, 1-24, Elsevier.

[4] Constable, R. L. and Howe, D. J., [1990], "Implementing Metamathematics as an Approach to Automatic Theorem Proving", in *Formal Techniques in Artificial Intelligence*, R. Banerji, ed., North-Holland.

[5] Constable, R. L., et al [1986], *Implementing Mathematics with the NUPRL Development System*, Prentice-Hall, N.J.

[6] Constable, R. L. [1983], "Programs as proofs", Information Processing Letters, 16(3), 105-112.

[7] Coquand, T. [1990], "On the analogy between propositions and types", in: *Logic Foundations of Functional Programming*, Addison-Wesley, Reading, MA.

[8] Coquand, T. and G. Huet [1985a], "Constructions: A higher order proof system for mechanizing mathematics", EUROCAL 85, Linz, Austria.

[9] Feferman, S. [1975], "A language and axioms for explicit mathematics", in: *Algebra and Logic*, Lecture Notes in Mathematics No. 450, pp. 87-139, Springer, Berlin.

[10] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, pages 61 – 80, Springer-Verlag, Argonne, IL, May 1988.

[11] Fitting, M. [1983], *Proof Methods for Modal and Intuitionistic Logics*, D. Reidel, Dordrecht, The Netherlands.

[12] Griffin, T [1990], "The Formulas-as-Types notion of control", in the proceedings of the 17th POPL conference.

[13] Harper, R., MacQueen, D., Milner, R., [1986], *Standard ML*, Technical Report, LFCS-86-2, Laboratory for the Foundations of Computer Science, University of Edinburgh.

[14] Hayashi, S.and H. Nakano [1989], *PX: A Computational Logic*, The MIT Press, Cambridge.

[15] Howard, W. A. [1980], "The Formulae-as-types notion of construction", in: Seldin, J.P. and J. R. Hindley (eds.), *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, New York, 479-490.

[16] Hyland, J. M. E. [1982], "The effective topos". in: Troelstra, A. S. and D. S. van Dalen (eds.), *L.E.J. Brouwer Centenary Symposium*, North-Holland, Amsterdam.

[17] Huet, G., [1990], ed. *Logic Foundations of Functional Programming Languages*, proceedings of a symposium from the Year of Programming at the University of Texas at Austin, 1986, Addison-Wesley, Reading, MA.

[18] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages*, 1987.

[19] Kleene, S. C. [1952] *Introduction to Metamathematics*, North Holland.

[20] Lipton, J., [1991], "Constructive Kripke Semantics and Realizability", in the proceedings of the *Logic for Computer Science* conference held at the Math. Sci. Research Institute, Berkeley, Nov. 1989.

[21] Lipton, J. [1992], "Relating Logic Programming and Propositions-as-Types", full version of this summary, to appear as a technical report, University of Pennsylvania.

[22] Lloyd, J. W., [1987], *Foundations of Logic Programming*, second ed., Springer-Verlag.

[23] Dale Miller and Gopalan Nadathur. λProlog Version 2.7. July 1988. Distribution in C-Prolog and Quintus sources.

[24] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. 1988. To appear in the Annals of Pure and Applied Logic.

[25] McCarty, D. C. [1986], "Realizability and recursive set theory", Annals of Pure and Applied Logic 32, 11-194.

[26] van Dalen, D. [1986], "Intuitionistic logic", in *The Handbook of Philosophical Logic*, Vol. III, 225-339, D. Reidel, Dordrecht.

[27] Martin-Löf, P. [1982], "Constructive Mathematics and Computer Programming", in Logic, Methodology and Philosophy of Science IV, North Holland, Amsterdam.

[28] Martin-Löf, P. [1984], *Intuitionistic Type Theory*, Studies in Proof Theory Lecture Notes, BIBLIOPOLIS, Napoli, Italy.

[29] Mitchell, J. and E. Moggi [1987], "Kripke-style models for typed lambda calculus", *Proceedings from Symposium on Logic in Computer Science*, Cornell University, June 1987, IEEE, Washington, D.C.

[30] Moggi, E. [1989] "Computational Lambda Calculus and Monads" in Proc. 4th IEEE Symposium on Logic in Computer Science, Asilomar.

[31] Moggi, E. [1990] "Notions of computation and monads". technical report, University of Edinburgh.

[32] Murthy, C., [1990], "Extracting Constructive Content from Classical Proofs: Compilation and the Foundations of Mathematics", Ph. D. dissertation, Cornell University.

[33] Murthy, C., [1990], "An Evaluation Semantics for Classical Proofs" 1991 LICS Proceedings.

[34] Nerode, A, [1956], *Composita, Equations, and Recursive Definitions*, Ph. D. Dissertation, University of Chicago.

[35] Nerode, A. and Shore, R. [1992], *Logic and Logic Programming*, to appear.

[36] Odifreddi, G. [1989], *Classical Recursion Theory*, North-Holland, Amsterdam.

[37] Robinson, J. A. [1965], "A machine-oriented logic based on the resolution principle", Journal of the ACM, V.12, 23-41.

[38] Statman, R., [1982], "Completeness, Invariance and lambda-definability", JSL.

[39] Tronci, E, [1991], "Equational Programming in Lambda Calculus", 6th LICS proceedings.

[40] Troelstra, A. S. and D. van Dalen [1988], *Constructivism in Mathematics: An Introduction*, Vol. II, Studies in Logic and the Foundations of Mathematics, Vol. 123, North-Holland, Amsterdam.

[41] Troelstra, A. S. [1973], *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Mathematical Lecture Notes 344, Springer-Verlag, Berlin.

Programming with Streams in Coq
A case study : the Sieve of Eratosthenes

François Leclerc*
CRIN URA CNRS 262
BP 239
54506 Vandoeuvre-les-Nancy cedex, France

Christine Paulin-Mohring†
LIP-IMAG URA CNRS 1398
ENS Lyon, 46 Allée d'Italie
69364 Lyon cedex 07, France

August 92

Abstract

A parallel dataflow program can be seen as a network of functional transformers over infinite lists (also called streams). G. Kahn and D. MacQueen [6, 7] proposed this paradigm in order to apply standard technics of semantics for proving the correctness of concurrent programs. On the other hand, type theory and the paradigm of proofs as programs has proved to be useful for the justification of functional programs. We investigate the programmation with infinite lists in strongly typed lambda-calculus by studying the example of the sieve of Eratosthenes. We develop this example both in system F and in the Calculus of Constructions where certified programs are extracted from constructive proofs. This study suggests suitable representations for the type of Streams in the system Coq [2].

1 Introduction

1.1 Programming with proofs

Usual programs involve inductive structures as natural numbers, lists or trees. We know quit well how to systematically represent these structures in strongly typed lambda-calculus as Girard's system F. We also know which induction principles are needed to reason about these programs in a system like Coq for instance.

An alternative possibility in Coq is to use realisability to extract correct programs from intuitionistic proofs of existential statements. From a proof of $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ the system Coq will extract a functional program f (written in a ML like language). We know from the metatheory that $\forall x.P(x) \Rightarrow Q(x, f(x))$ holds and consequently the program f is certified. Proving interactively $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ is like developing simultaneously the program and its proof of correctness.

*Francois.Leclerc@loria.fr
†cpaulin@lip.ens-lyon.fr

This research was partly supported by ESPRIT Basic Research Action "Logical Frameworks" and by MRT Programme de Recherche Coordonnées "Programmation avancée et Outils pour l'Intelligence Artificielle". It started during F. Leclerc's DEA practical period at LIP in spring 91.

1.2 Programming with coinductive types

The representation of coinductive types in strongly typed lambda-calculus has been less investigated. However an infinite list can be represented as a process for generating infinitely many elements. Such infinite lists, also called streams can be used to model concurrency in a functional way making the proofs of correctness easier to achieve.

A program which acts as a stream transformer can be written in a ML-like language with a lazy evaluation of constructors. In a language with full polymorphism like system F or the Calculus of Constructions, it is possible to internally represent the type of streams. We experiment these systems by applying the paradigm of proofs as programs to the construction of streams. This paper presents a complete development of various versions of the sieve of Eratosthenes in Coq.

1.3 Notations

Our aim is to give the main ideas for the constructions of the programs. We will not be too formal but the examples were developed completely formally in the Coq system.

We shall use terms either from the system F or from the Calculus of Constructions, with the following notations. Quantification is written as $\forall x : M.N$, implication is written $M \rightarrow N$ which associates to the right. The lambda-abstraction is written as $\lambda x : M.N$ or $\Lambda X : K.M$ for an abstraction over types variables, the application of term M to term N is written $(M\ N)$, it associates to the left, we shall sometimes omit the external parentheses. The type annotation after the ":" in quantification or abstraction will be omitted when it is clear from the context. We shall write $M \simeq N$ when the terms M and N are convertible.

The same names will be used for different constructions but in that case also the ambiguity can be solved from the context.

2 Representing Streams

In typed functional programming languages as ML, infinite lists are represented using a possible lazy evaluation of the arguments of the constructors. A value of type infinite list will be a pair with a value (the head of the list) as the first component and an arbitrary program of type infinite list (representing the tail of the list) as the second component. We can write a program which evaluates the first n elements of an infinite sequence. This program can possibly fail or loop.

It may seem impossible to represent streams in a strongly typed programming language where all programs are strongly terminating. However a possible concrete representation of lists in a language like system F will be as a process generating infinitely many objects.

We investigate possible representations of the type of streams in both system F and the Calculus of Constructions. The type of streams is a particular case of a coinductive type, namely a greatest fixpoint of a monotonic operator. We first recall the general scheme to represent coinductive types in system F.

2.1 Coinductive types in System F

The representation of inductive data types like integers or lists in system F can be seen as an encoding of smallest fixpoints of monotonic operators. A general presentation of this can be found in [1] or [4]. The case of coinductive types is less known but was studied by G. Wraith [11].

The representation of fixpoints of monotonic operators can be seen as an application of Tarski general fixpoint theorem. A coinductive type is the greatest fixpoint of a monotonic operator $A(X)$ which is computed as the least upper bound of the X such that $X < A(X)$. The order is just inclusion which is encoded via implication. The least upper bound is the union which is encoded via an existential type. This existential type is not a primitive notion in system F but can itself be encoded using a second-order quantification over types.

2.1.1 Basic definitions

In system F, the types in which a variable X occurs positively are generated by the entry Pos of the following syntax :

$$\begin{array}{lcl} Pos & ::= & M \mid X \mid \forall Y.Pos \mid Neg \to Pos \\ Neg & ::= & M \mid \forall Y.Neg \mid Pos \to Neg \end{array}$$

with the restriction that X does not occur in M.

Let $A(X)$ be a correct type in which X occurs positively, let Y be a type variable and f a variable of type $X \to Y$ then it is possible to build a term $A[f]$ of type $A(Y)$ by induction over $A(X)$.

Let $A(X)$ be a correct type of system F in which X occurs positively. We can define both μ and ν the smallest and the greatest fixpoint of this operator. We outline in the following table the main constructions associated to both types.

Inductive types	**Coinductive types**
$\mu \equiv \bigcap_X A(X) \subset X$ $\equiv \forall X.(A(X) \to X) \to X$	$\nu \equiv \bigcup_X X \subset A(X)$ $\equiv \exists X.(X \to A(X)) \wedge X$ $\equiv \forall C.(\forall X.(X \to A(X)) \to X \to C) \to C$
$\mathsf{iter} : \forall X.(A(X) \to X) \to \mu \to X$ $\equiv \Lambda X.\lambda f.\lambda m.(m\ X\ f)$ $\mathsf{in} : A(\mu) \to \mu$ $\equiv \lambda a.\Lambda X.\lambda f.(f\ (A[\mathsf{iter}\ X\ f]\ a))$	$\mathsf{build} : \forall X.(X \to A(X)) \to X \to \nu$ $\equiv \Lambda X.\lambda f.\lambda x.\Lambda C.\lambda H.(H\ X\ f\ x)$ $\mathsf{out} : \nu \to A(\nu)$ $\equiv \lambda m.(m\ A(\nu)\ \Lambda X.\lambda f.\lambda x.(A[\mathsf{build}\ X\ f]\ (f\ x)))$
$\mathsf{iter}\ X\ F\ (\mathsf{in}\ m) \to F\ (A[\mathsf{iter}\ X\ F]\ m)$	$\mathsf{out}\ (\mathsf{build}\ X\ F\ x) \to A[\mathsf{build}\ X\ F]\ (F\ x)$

2.1.2 Natural numbers and Streams

The typical example of an inductive definition is the type of natural numbers. It is the smallest fixpoint of the operator $1 + X$ which can be encoded in system F as $\forall C.C \to (X \to C) \to C$. The typical example of a coinductive definition is the type of streams of elements of a type A. It is the greatest fixpoint of the operator $A * X$ which can be encoded as $\forall C.(A \to X \to C) \to C$. In both cases, we can find more direct encoding than the one presented above. We outline the main constructions in the following table.

Natural numbers	**Streams**
$\mathsf{Nat} \leftrightarrow 1 + \mathsf{Nat}$ $\equiv \forall X.X \to (X \to X) \to X$	$\mathsf{Str} \leftrightarrow A * \mathsf{Str}$ $\equiv \exists X.(X \to A) \wedge (X \to X) \wedge X$ $\equiv \forall C.(\forall X.(X \to A) \to (X \to X) \to X \to C) \to C$
$\mathsf{iter} : \forall X.X \to (X \to X) \to \mathsf{Nat} \to X$ $\equiv \Lambda X.\lambda x.\lambda f.\lambda n.(n\ X\ x\ f)$ $0 : \mathsf{Nat}$ $\equiv \Lambda X.\lambda x.\lambda f.x$ $\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}$ $\equiv \lambda n.\Lambda X.\lambda x.\lambda f.(f\ (n\ X\ x\ f))$	$\mathsf{build} : \forall X.(X \to A) \to (X \to X) \to X \to \mathsf{Str}$ $\equiv \Lambda X.\lambda h.\lambda t.\lambda x.\Lambda C.\lambda H.(H\ X\ h\ t\ x)$ $\mathsf{hd} : \mathsf{Str} \to A$ $\equiv \lambda s.(s\ A\ \Lambda X.\lambda h.\lambda t.\lambda x.(h\ x))$ $\mathsf{tl} : \mathsf{Str} \to \mathsf{Str}$ $\equiv \lambda s.(s\ \mathsf{Str}\ \Lambda X.\lambda h.\lambda t.\lambda x.(\mathsf{build}\ X\ h\ t\ (t\ x)))$
$g : T \qquad h : T \to T$ $f \ :\ \mathsf{Nat} \to T \ \equiv\ (\mathsf{iter}\ T\ g\ h)$ $f\ 0 \ \simeq\ g$ $f\ (\mathsf{S}\ n) \ \simeq\ (h\ (f\ n))$	$g : T \to A \qquad h : T \to T$ $f \ :\ T \to \mathsf{Str} \ \equiv\ (\mathsf{build}\ T\ g\ h)$ $\mathsf{hd}\ (f\ x) \ \simeq\ (g\ x)$ $\mathsf{tl}\ (f\ x) \ \simeq\ f\ (h\ x)$

2.1.3 Examples

The nth element of a stream. To compute the nth element of a stream is done by induction on n. We define a function nth which has type $\mathsf{Nat} \to \mathsf{Str} \to A$. We want it to satisfy the following equations :

$$\mathsf{nth}\ 0\ s \simeq \mathsf{hd}\ s \qquad\qquad \mathsf{nth}\ (\mathsf{S}\ n)\ s \simeq \mathsf{nth}\ n\ (\mathsf{tl}\ s)$$

This is achieved by the following definition :

$$\mathsf{nth} \equiv (\mathsf{iter}\ \mathsf{Str} \to A\ \mathsf{hd}\ \lambda H : \mathsf{Str} \to A.\lambda s : \mathsf{Str}.(H\ (\mathsf{tl}\ s)))$$

This is an example of an iterative definition over an higher-order type ($\mathsf{Str} \to A$).

Stream of natural numbers starting from n. We can represent the stream of natural numbers ($n\ n+1\ n+2\ldots$). We define a function Enu which has type $\mathsf{Nat} \to \mathsf{Str}$. We want it to satisfy the following equations :

$$\mathsf{hd}\ (\mathsf{Enu}\ n) \simeq n \qquad \mathsf{tl}\ (\mathsf{Enu}\ n) \simeq (\mathsf{Enu}\ (\mathsf{S}\ n))$$

This is achieved with the definition : $\mathsf{Enu} \equiv (\mathsf{build}\ \mathsf{Nat}\ \lambda x.x\ \mathsf{S})$.

Following the same pattern we can represent the constant stream which only contains the value n as :

$$\mathsf{Con} \equiv (\mathsf{build}\ \mathsf{One}\ \lambda x.n\ \lambda x.x\ \mathsf{o})$$

with One a type (for instance $\forall X.X \to X$) with at least one element o.

2.1.4 Properties of streams

A "canonical element" of type streams will be a quadruple (X, H, T, x) with X a type, H a function of type $X \to A$, T a function of type $X \to X$ and x a term of type X. Such a stream is obtained by $(\mathsf{build}\ X\ H\ T\ x)$. Its head will be $(H\ x)$ and its tail a quadruple $(X, H, T, (T\ x))$. The nth element of this stream is $(H\ (T^n\ x))$ where $(T^0\ x) = x$ and $(T^{n+1}\ x) = (T^n\ (T\ x))$.

The stream (X, H, T, x) can be seen as an abstract representation of a process build on a type X, a current state x and functions H and T that computes the output of the stream and the modification of the state.

We can say that this representation is abstract because when we have an element of type stream, we cannot access its particular implementation.

Of course an infinite list can be implemented as many different objects of type streams. For instance, if s is a stream then $(\mathsf{build}\ \mathsf{Str}\ \mathsf{hd}\ \mathsf{tl}\ s)$ and $(\mathsf{build}\ \mathsf{Nat}\ \lambda n : \mathsf{Nat}.(\mathsf{nth}\ n\ s)\ S)$ are also streams which produce the same elements.

A natural question to ask is whether a close object of type Str is convertible to a term of the form $(\mathsf{build}\ X\ H\ T\ x)$. This is not true because of the impredicative representation of the existential type. For instance the constant stream can be represented as :

$$\begin{array}{l}\Lambda C.\lambda H : \forall X.(X \to \mathsf{Nat}) \to (X \to X) \to X \to C.\\ \qquad (H\ C \to C\ \lambda x : C \to C.n\ \lambda x : C \to C.x\ \lambda x : C.x)\end{array}$$

This sort of problem was also mentioned for inductive types in [10] although the representation of inductive types is adequate for some "simple" types as natural numbers. This problem will disappear in the system Coq if we use the general primitive notion of so-called inductive definitions (but which can be used also for non-inductive concrete definitions as existential, pairs or disjunctions). With this encoding of the existential type, a close normal term of type $\exists X.A(X)$ will be a pair (X, H) with a close type X and a close term H of type $A(X)$.

Iterative versus primitive recursive definitions. The impredicative encoding of natural numbers gives naturally the definition of functions using an iterative scheme $(f\ (\mathsf{S}\ n)) = (h\ (f\ n))$. The more general scheme of primitive recursive definition $(f\ (\mathsf{S}\ n)) = (h\ n\ (f\ n))$ is obtained using iteration and pairing. It is well known that this encoding is not satisfactory because $(f\ (\mathsf{S}\ n))$ reduces to $(h\ n\ (f\ n))$ only if n is a close natural numbers and also the number of reduction steps will be proportional to n. There is of course an analogous problem for streams. The basic iterative scheme of definition is $(\mathsf{tl}\ (f\ x)) = (f\ (h\ x))$. With this only scheme, it is not direct to program a function for the concatenation of an element a at the head of a stream s. The equations for such a definition will be :

$$(\mathsf{hd}\ (\mathsf{conc}\ a\ s)) = a \qquad\qquad (\mathsf{tl}\ (\mathsf{conc}\ a\ s)) = s$$

More generally we would like to program a stream as some process (X, h, t, x) which at some step becomes a new given stream. We write $X + \mathsf{Str} \equiv \forall C.(X \to C) \to (\mathsf{Str} \to C) \to C$ the type representing the disjunct union of X and Str. We call inl (resp. inr) the left (resp. right) injection of type $X \to X + \mathsf{Str}$ (resp. $\mathsf{Str} \to X + \mathsf{Str}$). The analogous of primitive recursion will be an operation build_pr of type :

$$\forall X.(X \to A) \to (X \to (X + \mathsf{Str})) \to X \to \mathsf{Str}.$$

We would like the following equalities to hold :

$$\begin{array}{rcll}\mathsf{hd}\ (\mathsf{build\_pr}\ X\ h\ t\ x) & = & (h\ x) & \\ \mathsf{tl}\ (\mathsf{build\_pr}\ X\ h\ t\ x) & = & \mathsf{build\_pr}\ X\ h\ t\ y & \text{if } (t\ x) = (\mathsf{inl}\ y)\\ \mathsf{tl}\ (\mathsf{build\_pr}\ X\ h\ t\ x) & = & s & \text{if } (t\ x) = (\mathsf{inr}\ s)\end{array}$$

If f is of type $X \to Y$ and g of type $\mathsf{Str} \to Y$, we write $[f, g]$ the function of type $(X + \mathsf{Str}) \to Y$ such that $[f, g]$ $(\mathsf{inl}\ x) \simeq (f\ x)$ and $[f, g]$ $(\mathsf{inr}\ y) \simeq (g\ y)$. We can mimic the behavior of build_pr by taking :

$$\begin{aligned}\mathsf{build\_pr} \equiv\ & \Lambda X.\lambda h : X \to A.\lambda t : X \to (X + \mathsf{Str}).\lambda x : X.\\ & (\mathsf{build}\ X + \mathsf{Str}\ [h, \mathsf{hd}]\ [t, \lambda s : \mathsf{Str}.(\mathsf{inr}\ (\mathsf{tl}\ s))]\ (\mathsf{inl}\ x))\end{aligned}$$

The equalities hd $(\mathsf{build\_pr}\ X\ h\ t\ x) = (h\ x)$ and tl $(\mathsf{build\_pr}\ X\ h\ t\ x) = (\mathsf{build\_pr}\ X\ h\ t\ y)$ if $(t\ x) = (\mathsf{inl}\ y)$ will be satisfied as convertibility rules. But if $(t\ x) = (\mathsf{inr}\ s)$, the streams tl $(\mathsf{build\_pr}\ X\ h\ t\ x)$ and s will not be convertible but will only have the same behavior.

Using build_pr, the stream $(\mathsf{conc}\ a\ s)$ can be defined as :

$$(\mathsf{build\_pr}\ One\ \lambda x.a\ \lambda x.(\mathsf{inr}\ s)\ o)$$

The definition of the analogous of primitive recursion for coinductive types in typed lambda-calculus is presented in [8, 3]. We will not need it in our development of the sieve of Eratosthenes.

Termination properties. In system F, every program terminates. If s is a stream, it implies that for each n, $\mathsf{nth}\ n\ s$ has a value. So an element of type Str define a total function from Nat to A.

2.2 Streams in the system Coq

The system Coq extends the Calculus of Constructions with a primitive mechanism for inductive definitions. It contains system F and the above representation of Streams can consequently be used. We do not use the extension with primitive inductive definitions for the encoding of streams but we shall use it for the type of natural numbers in order to make available the corresponding induction principle.

The Calculus of Constructions contains dependent types like equality or the existential type. This allows to reason about these streams. For instance we may state and prove properties like :

$$\forall n.(\mathsf{nth}\ n\ (\mathsf{Enu}\ 0)) = n$$

this will be done by proving more generally by induction over n :

$$\forall n, m : \mathsf{Nat}.(\mathsf{nth}\ n\ (\mathsf{Enu}\ m)) = n + m$$

But this is an external point of view where we first write a program (in that case a stream) and then prove its properties. We are looking for a method where we build at the same time the stream and its proof of correctness. One first possibility is to use for the type A of the elements of the stream a type which contains some logical information. For instance the constant stream with only n which was specified as a stream with elements of type Nat can in the Calculus of Constructions be specified as a stream with elements in the type $\exists p : \mathsf{Nat}.p = n$ of natural numbers which are equal to n. But this is clear that this kind of constant specification will not be very useful and it seems necessary to be able to parameterize the specification depending on the position of the element in the stream. We shall investigate a new definition of the type of streams with a type of output depending on the range of the element in the stream.

2.2.1 Indexed Streams

We can define a type for streams using dependent types that will allow an internal parameterization of streams with the index. A stream in system F gives for each n an element in a type A. Using dependent types, we can more generally in the Calculus of Constructions define a new type of streams that will give for each n a proof of a property $(A\ n)$. The main constructions related to such a stream are listed in the following table.

$$
\begin{array}{|rcl|}
\hline
\mathsf{Str} & \equiv & \lambda n:\mathsf{Nat}.\exists P:\mathsf{Nat}\to \mathit{Set}.(\forall m.(P\ m)\to(A\ m))\wedge(\forall m.(P\ m)\to(P\ (\mathsf{S}\ m)))\wedge(P\ n)\\
 & : & \mathsf{Nat}\to \mathit{Set}\\
 & \equiv & \lambda n:\mathsf{Nat}.\forall C:\mathit{Set}.\\
 & & (\forall P:\mathsf{Nat}\to \mathit{Set}.(\forall m.(P\ m)\to(A\ m))\to(\forall m.(P\ m)\to(P\ (\mathsf{S}\ m)))\to(P\ n)\to C)\\
 & & \to C\\
\hline
\mathsf{build} & : & \forall n:\mathsf{Nat}.\forall P:\mathsf{Nat}\to \mathit{Set}.\\
 & & (\forall m.(P\ m)\to(A\ m))\to(\forall m.(P\ m)\to(P\ (\mathsf{S}\ m)))\to(P\ n)\to(\mathsf{Str}\ n)\\
 & \equiv & \lambda n.\Lambda P.\lambda h.\lambda t.\lambda x.\Lambda C.\lambda H.(H\ \ P\ \ h\ t\ x)\\
\hline
\mathsf{hd} & : & \forall m.(\mathsf{Str}\ m)\to(A\ m)\\
 & \equiv & \lambda m.\lambda s.(s\ (A\ m)\ \Lambda P.\lambda h.\lambda t.\lambda x.(h\ m\ x))\\
\mathsf{tl} & : & \forall m.(\mathsf{Str}\ m)\to(\mathsf{Str}\ (S\ m))\\
 & \equiv & \lambda m.\lambda s.(s\ (\mathsf{Str}\ (\mathsf{S}\ m))\ \Lambda P.\lambda h.\lambda t.\lambda x.(\mathsf{build}\ (\mathsf{S}\ m)\ P\ h\ t\ (t\ m\ x)))\\
\hline
\end{array}
$$

The functions defined above satisfy the following convertibility rules with m, P, h, t and x of the appropriate types :

$$
\begin{array}{rcl}
\mathsf{hd}\ m\ (\mathsf{build}\ m\ P\ h\ t\ x) & \simeq & h\ m\ x\\
\mathsf{tl}\ m\ (\mathsf{build}\ m\ P\ h\ t\ x) & \simeq & \mathsf{build}\ (\mathsf{S}\ m)\ P\ h\ t\ (t\ m\ x)
\end{array}
$$

We can now prove the following property by induction over n.

$$\forall n.\forall m:\mathsf{Nat}.(\mathsf{Str}\ m)\to(A\ (n+m))$$

We call this proof nth. We can check that its computational behavior is similar to the one of the nth function in system F.

2.2.2 Examples

The example of the stream which contains successive numbers can be seen as an indexed stream. The specification of the output will be that for each n it gives a number p equal to n. This is represented in Coq as $\lambda n:\mathsf{Nat}.\exists p:\mathsf{Nat}.p=n$. Let us call A this specification. Because there exists obvious proofs h of $\forall m.(A\ m)\to(A\ m)$, t of $\forall m.(A\ m)\to(A\ (\mathsf{S}\ m))$, and for each n an object x of type $\exists p:\mathsf{Nat}.p=n$, we can define a stream on the specification A indexed by n as :

$$(\mathsf{build}\ n\ A\ h\ t\ x)$$

2.2.3 Computational interpretation

There exists a computational interpretation of the proofs of the Calculus of Constructions as programs of F_ω which is described in [9, 10]. If we perform this interpretation on indexed streams, we do not endup with the system F representation of streams but with a slightly heaviest representation :

$$\exists X.(\mathsf{Nat} \to X \to A) \wedge (\mathsf{Nat} \to X \to X) \wedge X$$

More or less this representation of streams contains an extra state of type Nat with the index stored in it.

We may avoid this problem if we introduce an extra notation for variables that will not be used for computations. These notations are $\lambda\!\!\lambda x.t$ and $\forall\!\!\!\forall x.P$. The precise rules and interpretation for this extension are in the spirit of what Hayashi did in the system ATT [5]. They have still to be written and are not the purpose of this paper but intuitively the ideas are the following :

The extraction of $\lambda\!\!\lambda x.t$ is just the extraction of t, if t is of type $\forall\!\!\!\forall x.P$ then the extraction of $(t\ u)$ is the extraction of t. A program p will be correct with respect to the specification $\forall\!\!\!\forall x.P$ if for all x, the program p is correct for P (usually we say that p is correct with respect to $\forall x.P$ if for all x, the result of the program p applied to x is correct for P.) We can statically check that a variable x is not used in a computational part which makes sure that the extraction process is correct.

With the following definitions for the type of indexed streams, we shall have as extracted terms exactly the system F analogous constructions.

$$\begin{array}{rcl}
\mathsf{Str} & \equiv & \lambda n : \mathsf{Nat}.\exists P : \mathsf{Nat} \to \mathit{Set}.(\forall\!\!\!\forall m.(P\ m) \to (A\ m)) \wedge (\forall\!\!\!\forall m.(P\ m) \to (P\ (\mathsf{S}\ m))) \wedge (P\ n) \\
 & : & \mathsf{Nat} \to \mathit{Set} \\
 & \equiv & \lambda n : \mathsf{Nat}.\forall C : \mathit{Set}. \\
 & & (\forall P : \mathsf{Nat} \to \mathit{Set}.(\forall\!\!\!\forall m.(P\ m) \to (A\ m)) \to (\forall\!\!\!\forall m.(P\ m) \to (P\ (\mathsf{S}\ m))) \to (P\ n) \to C) \\
 & & \to C
\end{array}$$

$$\begin{array}{rcl}
\mathsf{build} & : & \forall\!\!\!\forall n : \mathsf{Nat}.\forall P : \mathsf{Nat} \to \mathit{Set}. \\
 & & (\forall\!\!\!\forall m.(P\ m) \to (A\ m)) \to (\forall\!\!\!\forall m.(P\ m) \to (P\ (\mathsf{S}\ m))) \to (P\ n) \to (\mathsf{Str}\ n) \\
 & \equiv & \lambda\!\!\lambda n.\Lambda P.\lambda h.\lambda t.\lambda x.\Lambda C.\lambda H.(H\ \ P\ \ h\ t\ x)
\end{array}$$

$$\begin{array}{rcl}
\mathsf{hd} & : & \forall\!\!\!\forall m.(\mathsf{Str}\ m) \to (A\ m) \\
 & \equiv & \lambda\!\!\lambda m.\lambda s.(s\ (A\ m)\ \Lambda P.\lambda h.\lambda t.\lambda x.(h\ m\ x)) \\
\mathsf{tl} & : & \forall\!\!\!\forall m.(\mathsf{Str}\ m) \to (\mathsf{Str}\ (S\ m)) \\
 & \equiv & \lambda\!\!\lambda m.\lambda s.(s\ (\mathsf{Str}\ (\mathsf{S}\ m))\ \Lambda P.\lambda h.\lambda t.\lambda x.(\mathsf{build}\ (\mathsf{S}\ m)\ P\ h\ t\ (t\ m\ x)))
\end{array}$$

2.3 General parameterized Streams

In general the stream can be parameterized by an arbitrary sequence of objects in a given type U. The specification of the output will be a relation on U namely A of type $U \to U \to \mathit{Set}$. A stream parameterized by p of type U will give as an output a new parameter q, a proof of $(A\ p\ q)$ and its tail part will be a stream parameterized with q.

We do not give all the precise definitions for such a type of streams. It will be done in a particular case for the development of the partial version of the sieve of Eratosthenes.

3 The sieve of Eratosthenes in system F

We are now interested in the development of the sieve of Eratosthenes in system F.

3.1 Description of the algorithm

The algorithm is simple, we describe it equationally.

An auxiliary operation is the sieve function itself which takes a number p and a stream s and gives back the stream of the elements of s which cannot be divided by p. The operation sieve has type $\mathsf{Nat} \to \mathsf{Str} \to \mathsf{Str}$. It can be described by the following equation :

$$\begin{array}{rl}(\mathsf{sieve}\ p\ s) = & \mathsf{if\ div}\ p\ (\mathsf{hd}\ s)\ \mathsf{then\ sieve}\ p\ (\mathsf{tl}\ s) \\ & \mathsf{else\ conc}\ (\mathsf{hd}\ s)\ (\mathsf{sieve}\ p\ (\mathsf{tl}\ s))\end{array}$$

The sieve is used in an operation of type $\mathsf{Str} \to \mathsf{Str}$ that we call primes. This operation takes a stream s, keep its head p and applies to the tail of this stream the sieve operation using p. The primes function is characterized by the following equations.

$$\mathsf{hd}\ (\mathsf{primes}\ s) = (\mathsf{hd}\ s) \qquad \mathsf{tl}\ (\mathsf{primes}\ s) = \mathsf{primes}\ (\mathsf{sieve}\ (\mathsf{hd}\ s)\ (\mathsf{tl}\ s)))$$

The primes numbers are obtained by applying the primes operation to the stream (Enu 2).

If the scheme for the definition of the stream of prime numbers follows the general pattern of an iterative definition, it is not the case for the sieve function. The head of the stream (sieve s) will be the first element of s which can be divided by p. Nothing in the algorithm makes sure that this search terminates. Starting from a different stream than (Enu 2) we could loop forever. To say that the result of the sieve has type Str implies in particular that there are infinitely many primes numbers. It is not surprising that we cannot just translate this algorithm in system F.

3.2 A total version of the sieve

There are two solutions to avoid this problem. The first one is to introduce an explicit bound (from mathematical results we know how to compute one) for the search of the next number which cannot be divided by p. The second one is not to manipulate streams of natural numbers but streams of a type $\mathsf{Nat+}$ which contains an extra dummy element dum. The output will be a stream with primes numbers separated by the dummy element. Nothing in the type of the output avoids that, from one point, every output will be dummy.

We call this last algorithm, the total version of the sieve of Eratosthenes. We can program in system F a function $div+$ of type $\mathsf{Nat+} \to \mathsf{Nat+} \to \mathsf{bool}$, such that $(\mathsf{div+}\ \ p\ q) = \mathsf{true}$ when p or q is the dummy element or when p divides q (division on natural numbers is primitive recursive and can be programmed in system F). The operation sieve has type $\mathsf{Nat+} \to (\mathsf{Str\ Nat+}) \to (\mathsf{Str\ Nat+})$. It can be described by the following equations :

$$\begin{array}{rcl}\mathsf{hd}\ (\mathsf{sieve}\ p\ S) & = & \mathsf{if}\ (\mathsf{div+}\ \ p\ (\mathsf{hd}\ S))\ \mathsf{then\ dum}\ \mathit{else}\ (\mathsf{hd}\ S) \\ \mathsf{tl}\ (\mathsf{sieve}\ p\ S) & = & \mathsf{sieve}\ p\ (\mathsf{tl}\ S)\end{array}$$

This follows an iterative scheme. It is actually just the operation of applying a function f (in that case $\lambda q.\mathsf{if}\ (div+\ \ p\ q)\ \mathsf{then\ dum}\ \mathit{else}\ q$) to each element of a stream. It can easily be translated into exact code in system F.

$$\begin{array}{rl}\mathsf{map\_Str}: & (A \to B) \to (\mathsf{Str}\ A) \to (\mathsf{Str}\ B) \\ \equiv & \lambda f : A \to B.(\mathsf{build}\ (\mathsf{Str}\ A)\ \lambda s : (\mathsf{Str}\ A).(f\ (\mathsf{hd}\ s))\ \mathsf{tl})\end{array}$$

The primes operation has type $(\mathsf{Str\ Nat+}) \to (\mathsf{Str\ Nat+})$. The equations are not changed :

$$\mathsf{hd}\ (\mathsf{primes}\ s) = (\mathsf{hd}\ s) \qquad \mathsf{tl}\ (\mathsf{primes}\ s) = \mathsf{primes}\ (\mathsf{sieve}\ (\mathsf{hd}\ s)\ (\mathsf{tl}\ s)))$$

The primes numbers are obtained by applying the primes operation to the stream of successive elements in $\mathsf{Nat+}$ starting from 2 namely with inj the injection function from Nat to $\mathsf{Nat+}$.

$$\mathsf{Si} \equiv (\mathsf{build\ Nat\ inj\ S}\ 2)$$

We may remark that we will also get the stream of primes numbers starting from three by applying the primes operation to the stream of odd numbers starting from three. This stream can be constructed as :

$$(\mathsf{build\ Nat\ inj}\ \lambda n : \mathsf{Nat}.(\mathsf{S}\ (\mathsf{S}\ n))\ 3)$$

4 The sieve of Eratosthenes in Coq

We now want to have a proved version of the sieve of Eratosthenes, namely that it gives us all prime numbers.

Instead to do a proof of the previous program we shall build a new one that will give us as an output a natural number and a proof of its primality. Of course the problem will also be to specify that it gives us also all prime numbers.

We first study the proof counterpart of the total sieve which answers for each number if it is prime or not. We shall also gives the constructions corresponding to a stream which only contains the prime numbers.

4.1 Specification

If we analyze the algorithm, we can see that its specification does not concern really primality but can be parameterized by the initial stream the primes operation is applied to.

Let this stream be called Si. We shall denote $\mathsf{Si}\,(n)$ the n-th element of this stream. The output of the sieve program will be the elements of Si which cannot be divided by any of the previous elements in the stream. Of course if Si is the stream of natural numbers starting from two or the stream of odd numbers starting from three, the fact that they cannot be divided by a previous element is equivalent to the fact that they are prime.

This analysis led us to the definition of the relation divinf on natural numbers with the following meaning :

$$\mathsf{divinf}\ k\ n \equiv \exists p.(p < k \wedge div\ \mathsf{Si}\,(p)\ n)$$

A few logical lemmas about divinf will be used.

4.2 A total sieve development

We first develop a proved version of the sieve of Eratosthenes which corresponds from the computational viewpoint to the total sieve we studied in system F.

We introduce the predicate $\mathsf{divspec}$ of type $\mathsf{Nat} \to \mathsf{Nat} \to Set$. The definition of the specification ($\mathsf{divspec}\ k\ n$) is $(\exists p : \mathsf{Nat}.p = \mathsf{Si}\,(n) \wedge \neg(\mathsf{divinf}\ k\ p)) + (\mathsf{divinf}\ k\ \mathsf{Si}\,(n))$. The computational contents of this specification corresponds to the type $\mathsf{Nat+}$.

The output of the final sieve program will be an indexed stream based on the predicate $\lambda n : \mathsf{Nat}.(\mathsf{divspec}\ n\ n)$. It will indicate for the nth element of the stream whether there exists $k < n$ such that $\mathsf{Si}\,(k)$ divides $\mathsf{Si}\,(n)$.

The function sieve**.** The sieve transformation itself corresponds to a proof of :

$$\forall p : \mathsf{Nat}.\forall k : \mathsf{Nat}.(p = \mathsf{Si}\,(k)) \to \forall n.(\mathsf{Str}\ (\mathsf{divspec}\ k)\ n) \to (\mathsf{Str}\ (\mathsf{divspec}\ (\mathsf{S}\ k))\ n)$$

Let assume that we have p, k and a proof of $p = \mathsf{Si}\,(k)$. To prove $\forall n.(\mathsf{Str}\ (\mathsf{divspec}\ k)\ n) \to (\mathsf{Str}\ (\mathsf{divspec}\ (\mathsf{S}\ k))\ n)$ we use build with the predicate $(\mathsf{Str}\ (\mathsf{divspec}\ k))$ we have to prove the two following lemmas :

$$\forall n.(\mathsf{Str}\ (\mathsf{divspec}\ k)\ n) \to (\mathsf{divspec}\ (\mathsf{S}\ k)\ n)$$

and

$$\forall n.(\mathsf{Str}\ (\mathsf{divspec}\ k)\ n) \to (\mathsf{Str}\ (\mathsf{divspec}\ k)\ (\mathsf{S}\ n))$$

This last lemma is just proved by the tl term. For the first one, assuming we have n and an element s in $(\mathsf{Str}\ (\mathsf{divspec}\ k)\ n)$ we look at the head of the stream. It gives us a proof of $(\mathsf{divspec}\ k\ n) \equiv (\exists q : \mathsf{Nat}.q = \mathsf{Si}\,(n) \land \neg(\mathsf{divinf}\ k\ q)) + (\mathsf{divinf}\ k\ \mathsf{Si}\,(n))$. We do a case analysis. If there is a proof of $(\mathsf{divinf}\ k\ \mathsf{Si}\,(n))$ then there exists a $l < k$ such that $\mathsf{Si}\,(l)$ divides $\mathsf{Si}\,(n)$ so a fortiori $(\mathsf{divinf}\ (\mathsf{S}\ k)\ \mathsf{Si}\,(n))$ is satisfied and then there is a proof $(\mathsf{divspec}\ (\mathsf{S}\ k)\ n)$. In the other case, let q be such that $q = \mathsf{Si}\,(n) \land \neg(\mathsf{divinf}\ k\ q)$, we test whether q can be divided by p. If p divides q then because $p = \mathsf{Si}\,(k)$ and $q = \mathsf{Si}\,(n)$ we conclude that $(\mathsf{divinf}\ (\mathsf{S}\ k)\ \mathsf{Si}\,(n))$ is satisfied and then $(\mathsf{divspec}\ (\mathsf{S}\ k)\ n)$. In the other case from $\neg(\mathsf{divinf}\ k\ q)$ and $\neg(div\ \mathsf{Si}\,(k)\ q)$ we conclude $\neg(\mathsf{divinf}\ (\mathsf{S}\ k)\ q)$ and finally $(\mathsf{divspec}\ (\mathsf{S}\ k)\ n)$. This ends the proof of the sieve.

The function primes**.** The primes function corresponds to a proof of :

$$\forall k : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{divspec}\ k)\ k) \to (\mathsf{Str}\ \lambda n : \mathsf{Nat}.(\mathsf{divspec}\ n\ n)\ k)$$

It means that starting from a stream indexed by k and which gives for each n if $\mathsf{Si}\,(n)$ can be divided by $\mathsf{Si}\,(m)$ for $m < k$ we can build a "diagonal" stream also indexed by k which gives for each n whether $\mathsf{Si}\,(n)$ can be divided by $\mathsf{Si}\,(m)$ for $m < n$.

In order to prove this we apply build to the predicate $\lambda n : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{divspec}\ n)\ n)$. We have to prove the two lemmas :

$$\forall n : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{divspec}\ n)\ n) \to (\mathsf{divspec}\ n\ n)$$

which is just proved by the hd function. We also have to prove :

$$\forall k : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{divspec}\ k)\ k) \to (\mathsf{Str}\ (\mathsf{divspec}\ (\mathsf{S}\ k))\ (\mathsf{S}\ k))$$

Let s be a term of type $(\mathsf{Str}\ (\mathsf{divspec}\ k)\ k)$ we take the head of this stream. It gives a proof of $(\mathsf{divspec}\ k\ k) \equiv (\exists p : \mathsf{Nat}.p = \mathsf{Si}\,(k) \land \neg(\mathsf{divinf}\ k\ p)) + (\mathsf{divinf}\ k\ \mathsf{Si}\,(k))$. We do a case analysis. Let p be such that $p = \mathsf{Si}\,(k)$ we apply the sieve program to p, k, $(\mathsf{S}\ k)$ and the tail of s which has type $(\mathsf{Str}\ (\mathsf{divspec}\ k)\ (Sk))$. We get the expected proof of $(\mathsf{Str}\ (\mathsf{divspec}\ (\mathsf{S}\ k))\ (Sk))$. If $(\mathsf{divinf}\ k\ \mathsf{Si}\,(k))$ we use the only non trivial lemma about divinf which says that if n cannot be divided by any $\mathsf{Si}\,(m)$ with $m < k$ then because $\mathsf{Si}\,(k)$ can be divided by $\mathsf{Si}\,(l)$ with $l < k$ we have that n cannot be divided by $\mathsf{Si}\,(k)$. So $(\mathsf{divinf}\ k\ \mathsf{Si}\,(k)) \to \neg(\mathsf{divinf}\ k\ n) \to \neg(\mathsf{divinf}\ (\mathsf{S}\ k)\ n)$.

So from $\exists q : \mathsf{Nat}.q = \mathsf{Si}\,(n) \land \neg(\mathsf{divinf}\ k\ q)$ we deduce $\exists q : \mathsf{Nat}.q = \mathsf{Si}\,(n) \land \neg(\mathsf{divinf}\ (\mathsf{S}\ k)\ q)$. And because we always have $(\mathsf{divinf}\ k\ n) \to (\mathsf{divinf}\ (\mathsf{S}\ k)\ n)$ we deduce $(\mathsf{divspec}\ k\ n) \to (\mathsf{divspec}\ (\mathsf{S}\ k)\ n)$. We finally can map this proof on the tail of s to get a proof of

$(\mathsf{Str}\ (\mathsf{divspec}\ (\mathsf{S}\ k))\ (\mathsf{S}\ k))$. We may check that this proof which looks a bit complicated acts like identity on streams.

Because we have trivially a proof of $\neg(\mathsf{divinf}\ 0\ n)$ we may build the initial stream as a proof of $(\mathsf{Str}\ (\mathsf{divspec}\ 0)\ 0)$.

This is done using build with the predicate $\lambda n : \mathsf{Nat}.\exists s : \mathsf{Str}.s = (\mathsf{tl}^n\ \mathsf{Si})$. We have to prove :

$$\forall n.(\exists s : \mathsf{Str}.s = (\mathsf{tl}^n\ \mathsf{Si})) \to (\mathsf{divspec}\ 0\ n)$$

and

$$\forall n.(\exists s : \mathsf{Str}.s = (\mathsf{tl}^n\ \mathsf{Si})) \to \exists s : \mathsf{Str}.s = (\mathsf{tl}^{(\mathsf{S}\ n)}\ \mathsf{Si})$$

Let s be such that $s = (\mathsf{tl}^n\ \mathsf{Si})$. For the first lemma, we take its head which is equal to $\mathsf{Si}\,(n)$. It gives us a proof of $\exists q : \mathsf{Nat}.q = \mathsf{Si}\,(n) \wedge \neg(\mathsf{divinf}\ 0\ q)$ and then a proof of $(\mathsf{divspec}\ 0\ n)$. For the second lemma we take the tail of s and check that $(\mathsf{tl}^{(\mathsf{S}\ n)}\ \mathsf{Si}) = (\mathsf{tl}\ (\mathsf{tl}^n\ \mathsf{Si}))$.

Final program. Applying the primes operation to this initial stream gives us a proof of :

$$(\mathsf{Str}\ \lambda n : \mathsf{Nat}.(\mathsf{divspec}\ n\ n)\ 0).$$

Using the nth function we have a proof of $\forall n.(\mathsf{divspec}\ n\ n)$.

One step that remains to be done is to prove arithmetical lemmas like :

$$(\forall p.\mathsf{Si}\,(p) = p + 2) \to \forall n.(\mathsf{divspec}\ n\ n) \to ((\mathsf{is\_prime}\ n + 2) + \neg(\mathsf{is\_prime}\ n + 2))$$

or

$$(\forall p.\mathsf{Si}\,(p) = 2p + 3) \to \forall n.(\mathsf{divspec}\ n\ n) \to ((\mathsf{is\_prime}\ 2n + 3) + \neg(\mathsf{is\_prime}\ 2n + 3))$$

4.3 A partial filter development

We are now interested in a program that will only gives as output the numbers that cannot be divided by previous elements.

4.3.1 Specification

It is not easy to specify this problem using the index of this number in the actual stream. A more natural specification will be to relate the output to its index in the initial stream Si and also to the index of the previous element.

For this we use a general parameterized type of streams which is defined as follow. Let A be a specification of type $\mathsf{Nat} \to \mathsf{Nat} \to Set$. We give the type of the build and the out function. In that case the out function gives for a stream parameterized by p, a new parameter q, its head which is a proof of $(A\ p\ q)$ and its tail which is a stream parameterized by $(\mathsf{S}\ q)$[1].

Str	$\equiv$	$\lambda p : \mathsf{Nat}.\exists P : \mathsf{Nat} \to Set.(\forall p.(P\ p) \to \exists q : \mathsf{Nat}.(A\ p\ q) \wedge (P\ (\mathsf{S}\ q))) \wedge (P\ p)$
	:	$\mathsf{Nat} \to Set$
build	:	$\forall p : \mathsf{Nat}.\forall P : \mathsf{Nat} \to Set.$ $(\forall p.(P\ p) \to \exists q : \mathsf{Nat}.(A\ p\ q) \wedge (P\ (\mathsf{S}\ q))) \to (P\ p) \to (\mathsf{Str}\ p)$
out	:	$\forall p.(\mathsf{Str}\ p) \to \exists q : \mathsf{Nat}.(A\ p\ q) \wedge (\mathsf{Str}\ (\mathsf{S}\ q))$

[1]The general case would have been to give the tail as a stream parameterized by q, but in our case q will alway be equal to $(\mathsf{S}\ q')$ and the specification can more naturally be written as a relation between p and q' than as a relation between p and q

We introduce the notation :

$$\mathsf{between} \equiv \lambda n, m : \mathsf{Nat}.\lambda P : \mathsf{Nat} \rightarrow Prop.(n \leq m) \wedge \forall x.(n \leq x < m) \rightarrow (P\ x)$$

($\mathsf{between}\ n\ m\ P$) means that $n \leq m$ and P holds for all x such that $n \leq x < m$ The specification we shall use is :

$$\mathsf{ASieve} \equiv \lambda k, p, q : \mathsf{Nat}.(\exists n : \mathsf{Nat}.n = \mathsf{Si}\,(q) \wedge \neg(\mathsf{divinf}\ k\ n)) \wedge (\mathsf{between}\ p\ q\ \lambda x.(\mathsf{divinf}\ k\ \mathsf{Si}\,(x)))$$

($\mathsf{ASieve}\ k\ p\ q$) means that we have a number equal to $\mathsf{Si}\,(q)$ which is the first element in Si from $\mathsf{Si}\,(p)$ that cannot be divided by a $\mathsf{Si}\,(l)$ for $l < k$.

4.3.2 Bounded search

We have to search for an element in the stream which satisfies some property. We know the existence of a bound N for the number of step in the search.

To program this, we use that some order is well-founded. The order depends on the bound N, it is defined as :

$$n \prec_N m \equiv m < n \wedge m < N$$

It is clear that there cannot be any infinite decreasing chain for $\prec_N$ because if $\ldots n_k \prec_N \ldots \prec_N n_1 \prec_N n_0$ we have :

$$n_0 < n_1 < \ldots < n_k < \ldots < N$$

We shall use a principle of well-founded induction namely the property $\mathsf{WF}\text{-}\prec_N$:

$$\forall P : \mathsf{Nat} \rightarrow Set.(\forall n.(\forall m.(m \prec_N n) \rightarrow (P\ m)) \rightarrow (P\ n)) \rightarrow \forall n.(P\ n)$$

This property can be proved directly by induction over the bound N. We shall use the fact that $\forall n, m : \mathsf{Nat}.\neg(m \prec_0 n)$ and $\forall N, n, m : \mathsf{Nat}.((\mathsf{S}\ m) \prec_{(\mathsf{S}\ N)} (\mathsf{S}\ n)) \rightarrow m \prec_N n$.

For the case $N = 0$, we assume we have F of type $\forall n.(\forall m.(m \prec_0 n) \rightarrow (P\ m)) \rightarrow (P\ n))$ and n of type Nat. We apply F to n, we have to prove $\forall m.(m \prec_0 n) \rightarrow (P\ m)$ which is done by absurdity of $m \prec_0 n$.

For the induction step we have a proof *HypN* of

$$\forall P : \mathsf{Nat} \rightarrow Set.(\forall n.(\forall m.(m \prec_N n) \rightarrow (P\ m)) \rightarrow (P\ n)) \rightarrow \forall n.(P\ n),$$

a predicate P of type $\mathsf{Nat} \rightarrow Set$, a proof F of $\forall n.(\forall m.(m \prec_{(\mathsf{S}\ N)} n) \rightarrow (P\ m)) \rightarrow (P\ n)$ and n of type Nat. We apply F to n, we have to prove $\forall m.(m \prec_{(\mathsf{S}\ n)} n) \rightarrow (P\ m)$. Let m be such that $m \prec_{(\mathsf{S}\ N)} n$, we know that $m = (\mathsf{S}\ m')$. We get the proof of $(P\ (\mathsf{S}\ m'))$ using the hypothesis *HypN* on the predicate $\lambda q.(P\ (\mathsf{S}\ q))$ and on m'. We have to prove :

$$\forall n.(\forall m.(m \prec_N n) \rightarrow (P\ (\mathsf{S}\ m))) \rightarrow (P\ (\mathsf{S}\ n))$$

let p be of type Nat and *Hp* be a proof of $\forall m.(m \prec_N p) \rightarrow (P\ (\mathsf{S}\ m))$. We build a proof of $(P\ (\mathsf{S}\ p))$ with F applied to $(\mathsf{S}\ p)$. We then have to prove $\forall m.(m \prec_{(\mathsf{S}\ N)} (\mathsf{S}\ p)) \rightarrow (P\ m)$. But $m \prec_{(\mathsf{S}\ N)} (\mathsf{S}\ p)$ implies $m = (\mathsf{S}\ q)$ and $q \prec_N p$. We apply *Hp* to q and get a proof of $(P\ (\mathsf{S}\ q))$ which is $(P\ m)$. This ends up the proof.

The computational content of this proof can be written as a program in a ML like language :

```
let rec Wfind = function
   0    -> fun  F n -> (F n (fun m -> error))
| S N   -> fun  F n ->
            (F n (fun m -> Wfind N
                                (fun p Hp -> F (S p) (fun m -> Hp (m-1)))
                                (m-1)))
```

This primitive recursive functional looks slightly more complicated than the general recursive program which do the same task :

```
let rec Wfrec F n = F n (Wfrec F)
```

This second program does not make use of the bound N for the search. In Coq, we can choose to use the direct proof of the induction principle $\prec_N$ and we will get a strongly normalizable program. We can also just prove that the order we use is well-founded and then use the principle :

$$\forall R.(\mathsf{well\_founded}\ R) \rightarrow \forall P : U \rightarrow \mathit{Set}.(\forall n.(\forall m.(R\ m\ n) \rightarrow (P\ m)) \rightarrow (P\ n)) \rightarrow \forall n.(P\ n)$$

This axiom is interpreted via a realisability interpretation as the program WFrec. It does not make use of the proof of the well-foundness of the relation.

4.3.3 The sieve program

To be able to write the sieve program we need more information on the stream Si, namely that for each k it contains infinitely many element that cannot be divided by $\mathsf{Si}\,(l)$ for $l < k$.

We add such an hypothesis : $\forall k, n : \mathsf{Nat}.\exists N : \mathsf{Nat}.\exists p : \mathsf{Nat}.((n \leq p < N) \wedge \neg(\mathsf{divinf}\ k\ \mathsf{Si}\,(p)))$. If we want a program written in a total language we need to explicitly give the bound N, if we only want to develop the program using general fixpoint this property can be non informative and will only be used for proving the well-foundness of the order.

The sieve program will be a proof of :

$$\begin{aligned}\forall n : \mathsf{Nat}.\forall p, q : \mathsf{Nat}.\ &(\mathsf{between}\ p\ q\ \lambda x.(\mathsf{divinf}\ p\ \mathsf{Si}\,(x))) \rightarrow n = \mathsf{Si}\,(q)\\ &\rightarrow \forall r : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ r) \rightarrow (\mathsf{Str}\ (\mathsf{ASieve}\ (\mathsf{S}\ q))\ r)\end{aligned}$$

In order to do that, we assume we have n, p and q natural numbers, and proofs of $n = \mathsf{Si}\,(q)$ and $(\mathsf{betweeen}\ p\ q\ \lambda x.(\mathsf{divinf}\ p\ \mathsf{Si}\,(x)))$. We apply build with the predicate $(\mathsf{Str}\ (\mathsf{ASieve}\ p)$. We have to prove :

$$\forall r.(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ r) \rightarrow \exists t : \mathsf{Nat}.(\mathsf{ASieve}\ (\mathsf{S}\ q)\ r\ t) \wedge (\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))$$

For this, we use the fact that there exists N such that $\exists x : \mathsf{Nat}.r \leq x < N \wedge \neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x))$ and prove

$$\begin{aligned}\forall r.\ &(\exists x : \mathsf{Nat}.r \leq x < N \wedge \neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x)))\\ &\rightarrow (\mathsf{Str}\ (\mathsf{ASieve}\ p)\ r) \rightarrow \exists t : \mathsf{Nat}.(\mathsf{ASieve}\ (\mathsf{S}\ q)\ r\ t) \wedge (\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))\end{aligned}$$

using the well-founded induction over the order $\prec_N$. Our induction hypothesis will be :

$$\begin{aligned}\forall u : \mathsf{Nat}.\ &(u \prec_N r) \rightarrow (\exists x : \mathsf{Nat}.u \leq x < N \wedge \neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x)))\\ &\rightarrow (\mathsf{Str}\ (\mathsf{ASieve}\ p)\ u) \rightarrow \exists t : \mathsf{Nat}.(\mathsf{ASieve}\ (\mathsf{S}\ q)\ u\ t) \wedge (\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))\end{aligned}$$

Now we assume $\exists x : \mathsf{Nat}.r \leq x < N \wedge \neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x))$. Let s be of type $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ r)$. We apply it an out step. It gives us a parameter t, proof of $(\mathsf{ASieve}\ p\ r\ t)$ and a stream stl of type $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))$. We do an elimination on the proof of $(\mathsf{ASieve}\ p\ r\ t)$. It gives us a natural number m and proofs of $m = \mathsf{Si}\,(t)$, $\neg(\mathsf{divinf}\ p\ m)$ and $(\mathsf{between}\ r\ t\ \lambda x.(\mathsf{divinf}\ p\ \mathsf{Si}\,(x)))$.

Now we look whether n divides m or not.

Case n divides m. If n divides m then we apply the induction hypothesis to $(\mathsf{S}\ t)$. It is easy to check from the hypotheses that :

$$\forall x.r \leq x < (\mathsf{S}\ t) \rightarrow (\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x))$$

Consequently from $\exists x : \mathsf{Nat}.r \leq x < N \wedge \neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x))$, we get

$$\exists x : \mathsf{Nat}.((\mathsf{S}\ t) \leq x < N) \wedge \neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x)).$$

We have to show that $(\mathsf{S}\ t) \prec_N r$. ¿From $(\mathsf{between}\ r\ t\ \lambda x.(\mathsf{divinf}\ p\ \mathsf{Si}\,(x)))$, we get $r \leq t$ and then $r < (\mathsf{S}\ t)$. Also it implies that $(\mathsf{S}\ t) < N$ because otherwise there will be no x between r and N such that $\neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x))$.

The proof of $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))$ will be just the stream stl (the tail of s). We got a new parameter u and proofs of $(\mathsf{ASieve}\ (\mathsf{S}\ q)\ (\mathsf{S}\ t)\ u)$ and $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ u))$. It is enough to show from this that the property $(\mathsf{ASieve}\ (\mathsf{S}\ q)\ r\ u)$ holds. The only problem is to show $(\mathsf{between}\ r\ u\ \lambda x.(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x)))$ but this is a simple consequence of $(\mathsf{between}\ r\ (\mathsf{S}\ t)\ \lambda x.(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x)))$, and $(\mathsf{between}\ (\mathsf{S}\ t)\ u\ \lambda x.(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x)))$ which comes from $(\mathsf{ASieve}\ (\mathsf{S}\ q)\ (\mathsf{S}\ t)\ u)$.

Case n does not divide m. If n does not divide m then the proof of :

$$\exists t : \mathsf{Nat}.(\mathsf{ASieve}\ (\mathsf{S}\ q)\ r\ t) \wedge (\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))$$

is done by checking that indeed t satisfies the expected properties.

The proof of $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ t))$ is the tail stream stl. To prove $(\mathsf{ASieve}\ (\mathsf{S}\ q)\ r\ t)$ we first remark that the property $(\mathsf{between}\ r\ t\ \lambda x.(\mathsf{divinf}\ (\mathsf{S}\ q)\ \mathsf{Si}\,(x))$ is an easy consequence of $(\mathsf{between}\ r\ t\ \lambda x.(\mathsf{divinf}\ p\ \mathsf{Si}\,(x))$ and the fact that $p \leq (\mathsf{S}\ q)$. Because $m = \mathsf{Si}\,(t)$, it is enough to prove $\neg(\mathsf{divinf}\ (\mathsf{S}\ q)\ m)$. We have a proof of $\neg(\mathsf{divinf}\ p\ m)$. Let us assume that there exists $a < (\mathsf{S}\ q)$ such that $\mathsf{Si}\,(a)$ divides m and show that it leads to a contradiction. We cannot have $a < p$ because $\neg(\mathsf{divinf}\ p\ m)$, $a = q$ is not possible because $\mathsf{Si}\,(q) = n$ does not divide m, we cannot have $p \leq a < q$ because $\forall x.p \leq x < q \rightarrow (\mathsf{divinf}\ p\ \mathsf{Si}\,(x))$ so we shall have $b < p$ such that $\mathsf{Si}\,(b)$ divides $\mathsf{Si}\,(a)$ and consequently m. This ends the proof of this case and of the sieve program.

4.3.4 The primes function

The primes function is now not really difficult to prove. Its specification is :

$$\forall k : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{ASieve}\ k)\ k) \rightarrow (\mathsf{Str}\ \lambda n, m : \mathsf{Nat}.(\mathsf{ASieve}\ n\ n\ m)\ k)$$

The proof is done by applying build with the predicate $\lambda k : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{ASieve}\ k)\ k)$. We have to show that :

$$\forall p : \mathsf{Nat}.(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ p) \rightarrow \exists q : \mathsf{Nat}.(\mathsf{ASieve}\ p\ p\ q) \wedge (\mathsf{Str}\ (\mathsf{ASieve}\ (\mathsf{S}\ q))\ (\mathsf{S}\ q))$$

So let s be of type $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ p)$ We apply an out step. It gives us q, a proof of $(\mathsf{ASieve}\ p\ p\ q)$ and the tail stream stl of type $(\mathsf{Str}\ (\mathsf{ASieve}\ p)\ (\mathsf{S}\ q))$. ¿From this we need to get a proof of $(\mathsf{Str}\ (\mathsf{ASieve}\ (\mathsf{S}\ q))\ (\mathsf{S}\ q))$. ¿From the proof of $(\mathsf{ASieve}\ p\ p\ q)$ we get n such that $n = \mathsf{Si}\,(q)$ and $(\mathsf{between}\ p\ q\ \lambda x.(\mathsf{divinf}\ p\ \mathsf{Si}\,(x)))$. So we can apply the sieve lemma to n, p and q also to $(\mathsf{S}\ q)$ and stl. We get the expected proof of $(\mathsf{Str}\ (\mathsf{ASieve}\ (\mathsf{S}\ q))\ (\mathsf{S}\ q))$.

4.3.5 The final step

As previously it is easy to build from Si a proof of $(\mathsf{Str}\ (\mathsf{ASieve}\ 0)\ 0)$ to which we apply the primes operation. Now if we have a proof of $(\mathsf{ASieve}\ p\ p\ q)$, it implies that $\mathsf{Si}\,(q)$ cannot be divided by $\mathsf{Si}\,(k)$ with $k < p$ and $\forall x.p \leq x < q \rightarrow (\mathsf{divinf}\ p\ \mathsf{Si}\,(x))$ this implies that $\mathsf{Si}\,(q)$ cannot be divided by $\mathsf{Si}\,(k)$ with $k < q$ which is the expected result for testing primality.

5 Conclusion

In this paper we investigated the programmation with infinite lists in strongly typed lambda-calculus by developing the example of the sieve of Eratosthenes. The complete proofs were developed in the system Coq.

It appears when we want to develop programs concerning streams that the methods to indicate in a specification which part will be used for computation is too weak in Coq. If we want to control exactly the computational aspect of the streams which are used, we shall need a method to indicate that a parameter of the problem should not appear as an input of the extracted program.

When trying to develop basic programs (for instance sorting algorithms) in the pure Calculus of Constructions axioms like induction over natural numbers were always needed. In the development of this algorithm on streams we did not need to add any axiom concerning coinductive types to the theory. The only axiom we used is the one concerning well-founded induction for the development of the partial recursive sieve. But it is not surprising because we know it is not possible to write in the Calculus of Constructions a proof of a search which does not make a computational use of a bound for this search.

The main difficulty to carry out this development was first to get the idea of using a parameterized type for streams, second to find the correct predicate to use when we apply the build operation. In general we know from the expected program what is the computational part of this predicate. For the logical part, it corresponds to find a correct invariant for the problem. This is a mathematical difficulty which often appears in the domain of proofs of programs. As soon as we get the right specifications and invariants, the system Coq is of a great help to carry out the whole formal development of the proof.

References

[1] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.

[2] G. Dowek et al. The Coq Proof Assistant User's Guide Version 5.6. Rapport Technique 134, INRIA, December 1991.

[3] H. Geuvers. Inductive and coinductive types with iteration and recursion. Faculty of Mathematics and Informatics, Catholic University Nijmegen, 1990.

[4] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[5] S. Hayashi. Singleton, union, intersection types for program extraction. In *Proceedings of TACS'91*, 1991.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*. North-Holland, 1974.

[7] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77*. North-Holland, 1977.

[8] N. Mendler. Predicative types universes and primitive recursion. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 173–184, Amsterdam, The Netherlands, 1991. IEEE Computer Society Press.

[9] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the Calculus of Constructions. In Association for Computing Machinery, editor, *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989.

[10] C. Paulin-Mohring. *Extraction de programmes dans¡ le Calcul des Constructions.* PhD thesis, Université Paris 7, January 1989.

[11] G. C. Wraith. A note on categorical data types. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*. Springer-Verlag, 1989. LNCS 389.

Compositional understanding of type theory

Zhaohui Luo, Edinburgh

Abstract

We present a theory of dependent types and discuss several issues concerning its conceptual universe of types, which include the relationship between data types and propositions, intensionality, and compositional understanding.

The new Implementation of ALF

Lena Magnusson*
Department of Computer Science,
University of Göteborg/Chalmers

Preliminary version, August 1992

1 Introduction

This paper describes the (completely) new implementation of ALF, which is an interactive proof editing system, still on the experimental stage. The aims are to have a general mechanism for adding and treating definitions in a uniform and powerful way, and to do proof editing as with "scratch paper and pencil". Theories are represented in the framework as collection of *constant declarations*, place holders are represented as *yet unknown* constant definitions and computation rules are defined by *pattern matching*. The flexibility in proof editing comes from the possibility to have several goals simultaneously, to combine top-down and bottom-up derivation, and to be able to backtrack in a structured way. Many of these features originate in the first ALF-system [ACN90] and an earlier implementation of Martin-Löf's logical framework [Mag91].

2 System Overview

The proof editing system is based on Martin-Löf's logical framework [NPS90], with the extension of judgements for contexts and substitutions (local environments) and the notion of explicit substitution applied to terms. The framework provides a general machinery for the theory independent part of the proof derivation system and the possibility of representing theories in the framework. Theories, definitions, proofs and incomplete derivations are all represented as (collections of) *constants* declared in an *environment*. Proof editing consists of manipulating the *scratch area* part of the environment, by adding definitions (new goals and lemmas), refining goals by combining rules from the theory, primitives and lemmas (proved or not), extending the theory with a completed proof, or deleting parts of a proof.

The system consists of three main parts, the environment with theory definitions and scratch area, the *checker* and the *command interpreter*. The latter will not be a subject of this paper, and we will refer instead to the system manual, which is under development. The different checking algorithms are all based on rules of the framework, and are parameterized over the environment definitions.

*lena@cs.chalmers.se

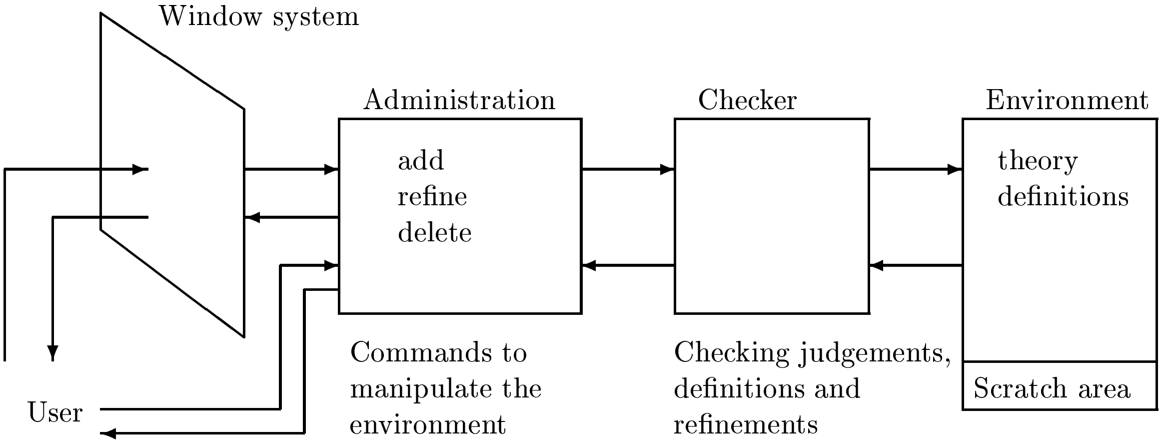

Figure 4: System overview

The framework provides general rules for application, abstraction, substitution and reduction (mainly beta and eta). A theory is represented by a collection of typed constants, where every "constant - type" pair corresponds to giving an axiom (state the proposition without proof). Rules, in a natural deduction style formulation of the theory, are represented as functions from premises to conclusions. Definitions of functions are given with pattern matching in a functional language style.

The environment consists of three parts, the theory, the dictionary and the scratch area. *Primitive* constants (given without justification) belong to the theory and *abbreviations* (which can be justified by type checking) are kept in the dictionary. All constants can be declared in a *local context*, which allows for defining theories and proofs parameterized over the context variables. These generalized (open) theories and proofs can be instantiated to particular (closed) theories and proofs by specializing the free variables of the constants with explicit substitutions.

The idea of the scratch area is that "unsure" parts of proofs can be manipulated in a similar way as on a "scratch paper". Once completed in a satisfactory way, the proof can be moved to the dictionary part of the environment. To manage backtracking and gluing parts together in a correct way, some book keeping is required. For instance a full *dependency graph* is kept and updated continuously, to prevent circular definitions. Moreover, a set of *constraints* are kept, which are requirements concerning unknown notions needed to be satisfied in the future refinements. All this book keeping is the reason for not allowing backtracking in the entire environment.

3 The framework

3.1 Terms and types

Terms in ALF are lambda terms, extended with a notation for explicit substitution and with constants declared in the environment. The syntax of terms is the following

$$e ::= \; x \mid c \mid [x_1, \ldots, x_n]e \mid e(e_1, \ldots, e_n) \mid c\{x_1 := e; \ldots; x_n := e\} \quad \text{for } n > 0,$$

where x denotes variables and c constants.
Additional restrictions are that 1) in an abstraction $[x_1, \ldots, x_n]e$, e is not itself an abstraction, 2) the head e in an application $e(e_1, \ldots, e_n)$ is not an application and 3) the variables $x_1, \ldots, x_n$ in an explicit substitution $\{x_1 := e; \ldots; x_n := e\}c$ must all be declared in the local context of the constant c.

Open terms contain free variables. These terms must be validated in a *context*, which is a sequence of typings of free variables

$$[x_1 : \alpha_1, x_2 : \alpha_2, \ldots, x_n : \alpha_n] \qquad (n \geq 0)$$

where x_i, $i = 1, \ldots, n$ are distinct variables, and the free variables occurring in α_i must have been declared earlier in the context. A substitution is a sequence of assignments of terms to variables

$$\{x_1 := a_1, x_2 := a_2, \ldots, x_n := a_n\}$$

The substitutions are performed *simultaneously*, instantiating (some of) the free variables of the term. Only open terms (or constants denoting open terms) can be affected by a substitution. The extension of syntax for explicit substitution is especially useful for instantiating variables of yet unknown constants, or for constructing generalized proofs (proofs in a context) which can later be specialized to particular (closed) proofs.

Types are generated from a set of *ground* types and a *dependent* function type constructor. A ground type is either the predefined type *Set* or a term of type *Set*. The syntax of ground types are

$$\alpha_{ground} ::= \ Set \mid e \qquad \text{(provided } e : Set\text{)}$$

and for types

$$\alpha ::= \ \alpha_{ground} \mid (\delta_1, \ldots, \delta_n)\alpha_{ground} \quad \text{for } n > 0,$$

where

$$\delta ::= \ x : \alpha \mid x_1, \ldots, x_k : \alpha \mid \alpha \quad \text{for } n > 0.$$

We will denote terms by a, b, e or A, B, types by α, β, contexts by Γ, Δ and substitutions by γ. An *extension* of a context Γ (or substitution γ) is denoted by $\Gamma.[x : \alpha]$ (or by $\gamma.\{x := a\}$).

3.2 The framework rules

The extensions compared to the original formulation of Martin-Löf's logical framework are the two judgements concerning contexts and substitutions and that judgements are given relative to contexts.

The basic assertions in the framework logic are the following judgement forms

Γ : Context	Γ is a context
$\gamma : \Delta \quad \Gamma$	γ is a substitution which fits context Δ in the context Γ
α : Type $\quad \Gamma$	α is a type in the context Γ
$\alpha = \beta : Type \quad \Gamma$	α and β are equal types in the context Γ
$a : \alpha \quad \Gamma$	a is an object of type α in the context Γ
$a = b : \alpha \quad \Gamma$	a and b are equal objects of type α in the context Γ

3.2.1 Context rules

A context is either empty or an extension of a proper context:

$$[\,] : Context \qquad\qquad \frac{\begin{array}{c}\Gamma : Context \\ \alpha : Type \quad \Gamma \\ x \notin \Gamma\end{array}}{\Gamma.[x : \alpha] : Context}$$

A substitution γ *fits* a context Δ if for every $x_i := a_i$ in γ, x_i is in the context Δ and a_i has the type of x_i. Also this relation is relative to a context (which will contain the free variables of the a_i's). This is also referred to as a *contextual mapping*, with the notation $\gamma : \Delta \rightarrow \Gamma$.

$$\{\,\} : [\,] \quad \Gamma \qquad\qquad \frac{\begin{array}{c}\gamma : \Delta \quad \Gamma \\ \alpha : Type \quad \Delta \\ a : \alpha\gamma \quad \Gamma\end{array}}{\gamma.\{x := a\} : \Delta.[x : \alpha] \quad \Gamma}$$

3.2.2 Term and Type rules

The new formation rules, concerning contexts are the following

$$\frac{\begin{array}{c}\gamma : \Delta \quad \Gamma \\ \alpha : Type \quad \Delta\end{array}}{\alpha\gamma : Type \quad \Gamma} \qquad\qquad \frac{\begin{array}{c}\gamma : \Delta \quad \Gamma \\ a : \alpha \quad \Delta\end{array}}{a\gamma : \alpha\gamma \quad \Gamma}$$

For the other rules, we will have to refer to [NPS90].

3.3 Representing a theory

There are mainly two different kinds of definitions, primitive and defined notions. As mentioned, primitive notions can not be justified inside the system, whereas defined notions can be justified by type checking.

3.4 Primitive notions

The primitive notions are divided into two groups, *constructors*, which could be either set constructors or element constructors, and *implicitly defined constants*. Implicit definitions are used for representing rules of theories such as structural induction, or for defining primitive functions on datatypes. These are called implicit since their meaning is given by (usually recursive) equations, which show their computational behavior.

Primitive constant definitions have the following form

$$name : \alpha \quad \Gamma \qquad \text{where name is a unique identifier}$$

such as $N : Set, 0 : N$ and $succ : (N)N$. Implicit definitions consist of an additional list of *pattern equations*, describing the computational behavior of the rule (function). Examples of implicitly defined constants are

$$
\begin{aligned}
add: \;& (N;N)N \\
& add(x,0) = x \\
& add(x, succ(y)) = succ(add(x,y))
\end{aligned}
$$

$$
\begin{aligned}
natrec: \;& (P:(N)Set; P(0); (x:N; P(x))P(succ(x)); n:N)P(n) \\
& natrec(P,d,e,0) = d \\
& natrec(P,d,e,succ(x)) = e(x, natrec(P,d,e,x))
\end{aligned}
$$

where patterns are either variables or element constructors (possibly applied to a substitution). The constructors must be applied to the proper number of arguments which are again patterns. A pattern can also be *redundant* (denoted _) which means that the argument is uniquely determined by type checking, and can therefore be safely omitted. These generalized patterns (compared to functional languages) come from the dependent types. Nonlinear patterns are one example of redundancy, but there are other cases as well. As a simple example, consider the first projection $fst : (A, B : Set; p : Prod(A,B))A$ where *pair* is the constructor of the product type $Prod$. The pattern equation should be

$$fst(\_,\_,pair(A,B,a,b)) = a$$

or equally

$$fst(A,B,pair(\_,\_,a,b)) = a$$

We require the pattern equations to be *non overlapping* (or mutually disjoint), which means that at most one pattern can match any term, and there is a way to ensure the pattern equations to be *exhaustive*, by using a built in feature for generating pattern equations. The pattern matching is described in detail in [Coqb].

3.5 Defined notions

Defined notions are either named terms together with their types and local contexts, or yet unknown notions. There is also a possibility of giving a name to a context or a substitution, and use the name as an abbreviation of the context or substitution, respectively. A yet unknown notion is simply a defined notion where the term is not yet known, but which is intended to be instantiated at a later point. Yet unknown notions are used for place holders. A constant definition is of the following form

$$c = a : \alpha \quad \Gamma$$

where a is a term which may contain free occurrences of the variables in context Γ, or

$$c = ? : \alpha \quad \Gamma$$

which denotes that c is yet unknown. The name c must be unique. Contexts can be concatenated (extended), denoted by $\Gamma_1 + \Gamma_2$. Context abbreviations are written as follows

$$name = \Gamma_1 + \cdots + \Gamma_n \qquad n > 0$$

where Γ_i can be either a context or a defined name. Substitution abbreviations are written

$$name = \gamma : \Delta \quad \Gamma$$

Substitutions can also be composed, with $\gamma_1 \circ \gamma_2 : \Gamma_1 \quad \Gamma_2$ if $\gamma_1 : \Gamma_1 \quad \Gamma$ and $\gamma_2 : \Gamma \quad \Gamma_2$. The abbreviations of contexts and substitutions are simply short hand notations, since whenever needed, the abbreviations are expanded to their definitions.

4 Judgement checking

Judgements usually depend on constants from the current theory and dictionary, which are declared in the environment (denoted Σ), and therefore the checking algorithm is parameterized over Σ. We are not explicitly stating the validity of the environment in the rules below, but it is assumed. the notion of valid environment is described in section 5.

The judgement checking algorithm will proceed in three steps, by checking 1) wellformedness, 2) validity and finally 3) truth of the judgement. The judgement forms presented above, are all decidable (with some minor restrictions), provided that the pattern equations given in the theory will give rise to a confluent rewriting system. This prerequisite is needed since each pattern equation will correspond to a reduction rule (from left to right), and the equality of terms is modulo all reduction equalities. Abbreviations are "reduced" by expansion of their definition, but these are harmless rules, concerning confluence. On the other hand, the incorporation of place holders as yet unknown constant definitions, have the effect that the equality of two terms can be *not yet decidable*, if the terms contain constants which have no definitions as yet. Therefore, the judgement checking algorithm will not always answer either yes or no, it will sometimes answer "maybe", meaning that the judgement might become true if the yet unknown constants are instantiated properly (fulfill the *constraints*). The second step in the algorithm concerns validity of a judgement, which is weaker than a true judgement since it means that the judgement will become true if the additional requirements of the constraints are satisfied. A judgement can be decided to be true or false as soon as there are no yet unknown constants involved (and sometimes refuted before that).

4.1 Constraints

A constraint is a tuple $(a_1, a_2, \alpha, \Gamma)$ in which at least one of the terms a_1 and a_2 are not *complete*, where complete means not depending on any yet unknown constants. The meaning of the constraint is that the unknown(s) in the equality is required to be instantiated in such a way that

$$a_1 = a_2 : \alpha \quad \Gamma$$

becomes a complete and true judgement.

The judgement checking algorithm produces a set of constraints, restricting the possible instantiations of the yet unknowns, or a special symbol $Fail$, denoting that the judgement can never become true, for any instantiations of the unknowns.

4.2 Valid judgements

The syntax of a wellformed judgement is as follows

$$J ::= \Gamma : Context \mid \gamma : \Delta \quad \Gamma \mid \alpha : Type \mid \alpha = \beta : Type \quad \Gamma \mid a : \alpha \quad \Gamma \mid a = b : \alpha \quad \Gamma$$

The validity of a judgement is computed by the function $\mathcal{J}$, which takes a judgement and computes a set of constraints (the union of all constraints resulting from the premises) or $Fail$ if any of the premises computes to $Fail$. J is defined in terms of $IsContext, FitsContext, IsType$

and $IsElem$ which are the constrained counterparts of the formation rules for contexts, substitutions, types and terms, and of the functions computing conversion of types and terms ($TypeConv$ and $Conv$). All constraints originate in the $Conv$ function, since its purpose is to check if two terms are convertible (equal modulo equality rules). First we give the rules of a valid judgement, and then state the additional requirements for a true judgement. We will also describe the conversion algorithm ($Conv$), but the other functions are directly corresponding to their respective formation rules, and are omitted.

$$\textbf{Context}: \ \frac{IsContext(\Sigma,\Gamma) \Rightarrow \xi}{\mathcal{J}(\Sigma, \Gamma : Context) \Rightarrow \xi} \qquad \textbf{Subst}: \ \frac{\begin{array}{c} IsContext(\Sigma,\Delta) \Rightarrow \xi_1 \\ IsContext(\Sigma,\Gamma) \Rightarrow \xi_2 \\ FitsContext(\Sigma,\Gamma,\Delta,\gamma) \Rightarrow \xi_3 \end{array}}{\mathcal{J}(\Sigma, \gamma : \Delta \quad \Gamma) \Rightarrow \xi_1 \cup \xi_2 \cup \xi_3}$$

$$\textbf{Type}: \ \frac{\begin{array}{c} IsContext(\Sigma,\Gamma) \Rightarrow \xi_1 \\ IsType(\Sigma,\Gamma,\alpha) \Rightarrow \xi_2 \end{array}}{\mathcal{J}(\Sigma, \alpha : Type \quad \Gamma) \Rightarrow \xi_1 \cup \xi_2} \qquad \textbf{TypeEq}: \ \frac{\begin{array}{c} IsContext(\Sigma,\Gamma) \Rightarrow \xi_1 \\ IsType(\Sigma,\Gamma,\alpha) \Rightarrow \xi_2 \\ IsType(\Sigma,\Gamma,\beta) \Rightarrow \xi_3 \\ TypeConv(\Sigma,\Gamma,\alpha,\beta) \Rightarrow \xi_4 \end{array}}{\mathcal{J}(\Sigma, \alpha = \beta : Type \quad \Gamma) \Rightarrow \bigcup_{i=1}^{4} \xi_i}$$

$$\textbf{Elem}: \ \frac{\begin{array}{c} IsContext(\Sigma,\Gamma) \Rightarrow \xi_1 \\ IsType(\Sigma,\Gamma,\alpha) \Rightarrow \xi_2 \\ IsElem(\Sigma,\Gamma,\alpha,a) \Rightarrow \xi_3 \end{array}}{\mathcal{J}(\Sigma, a : \alpha \quad \Gamma) \Rightarrow \xi_1 \cup \xi_2 \cup \xi_3} \qquad \textbf{ElemEq}: \ \frac{\begin{array}{c} IsContext(\Sigma,\Gamma) \Rightarrow \xi_1 \\ IsType(\Sigma,\Gamma,\alpha) \Rightarrow \xi_2 \\ IsElem(\Sigma,\Gamma,\alpha,a) \Rightarrow \xi_3 \\ IsElem(\Sigma,\Gamma,\alpha,b) \Rightarrow \xi_4 \\ Conv(\Sigma,\Gamma,\alpha,a,b) \Rightarrow \xi_5 \end{array}}{\mathcal{J}(\Sigma, a = b : \alpha \quad \Gamma) \Rightarrow \bigcup_{i=1}^{5} \xi_i}$$

Formally, we define a *true judgement* to be

$$\Gamma \vdash_\Sigma J \quad \text{iff} \quad J \text{ is complete and } \mathcal{J}(\Sigma, J) \Rightarrow \emptyset \qquad \text{(empty set of constraints)}$$

and conversely, a judgement is said to be *false* iff $\mathcal{J}(\Sigma, J) \Rightarrow Fail$. The notion of being *complete* is defined in more detail in section 6.1.

The first step of the conversion algorithm is to reduce the two terms to *head normal form*, and we will therefore proceed by giving the reduction rules, before the conversion is described.

4.3 Reduction rules

The reduction is performed by applying the one step reduction rules until one of the following cases occur

- the term is on head normal form (the *head* of the term is a constructor or a variable, or the term is an abstraction)
- the term is irreducible
- the term can not be reduced any further at this point, since there is a not yet defined constant blocking the reduction.

4.3.1 One step reduction rules

The β-reduction is defined by explicit substitution as an "uncurried" version of the usual β-rule, meaning that several bound variables are substituted simultaneously. The δ-expansion is simply unfolding a defined constant (with or without definition) and the last rule is matching against given pattern equalities.

β-reduction

$$([x_1,\ldots,x_n]b)(a_1,\ldots,a_k) \longrightarrow_\beta \begin{cases} b\{x_1 = a_1,\ldots,x_n = a_n\} & \text{if } n = k \\ [x_{k+1},\ldots,x_n](b\{x_1 = a_1,\ldots,x_k = a_k\}) & \text{if } k < n \\ (b\{x_1 = a_1,\ldots,x_n = a_n\})(a_{n+1},\ldots,a_k) & \text{if } k > n \end{cases}$$

δ-expansion

$$\frac{c = a : \alpha \quad \Gamma \in \Sigma}{c \longrightarrow_\delta a} \qquad\qquad \frac{c =? : \alpha \quad \Gamma \in \Sigma}{c \longrightarrow_\delta \mathit{Unknown}}$$

Pattern matching

Since we require the patterns to be non overlapping, we know that in the list of patterns associated to a given constant, in any position there must be either only pattern variables or only constructors. This means that if a yet unknown constant is found in a position of a constructor, we know that no other pattern can possibly match as yet and the term is *not yet reducible* and if a local variable is found in a constructor position, we know that no pattern can match at all, and the term is *irreducible*. Moreover, redundant patterns are only allowed in positions where type correctness guarantees a specific term, and therefore can safely be ignored in the matching. There are three cases

- if $c(p_1,\ldots,p_n) = a \in \Sigma$ and $\exists\gamma.c(p_1\gamma,\ldots,p_n\gamma) \equiv c(a_1,\ldots,a_n)$ then
$$c(a_1,\ldots,a_n) \longrightarrow_M a\gamma$$
- the pattern matching term is *not yet reducible*
- the pattern matching term is *irreducible*

4.3.2 Reduction to head normal form

The reason for not always reaching a term on head normal form is either that the term is $c(b_1,\ldots,b_n)$, where c is an implicitly defined constant and at some position i, where constructors are expected in the patterns, the b_i is either a variable or a yet unknown constant or the unknown is in the head position of the term. Therefore the output of the reduction will be a term prefixed with its status of reduction, i.e

$$\mathit{Term} \longrightarrow_{hnf} \mathit{Hnf(Term)} + \mathit{Irr(Term)} + \mathit{NotYet(Term)}$$

and the reduction proceeds by case analysis on the term structure

Var

$$x \longrightarrow_{hnf} \mathit{Hnf}(x)$$

Const

$c \longrightarrow_{hnf} \mathit{Hnf}(c)$ if c is a constructor
$c \longrightarrow_{hnf} \mathit{NotYet}(c)$ if c is yet unknown
$c \longrightarrow_{hnf} P(e')$ if $c \longrightarrow_{\delta} e \longrightarrow_{hnf} P(e')$, where $P \in \{\mathit{Hnf}, \mathit{Irr}, \mathit{NotYet}\}$

Abs

$[x_1, \ldots, x_n]a \longrightarrow_{hnf} \mathit{Hnf}([x_1, \ldots, x_n]a)$

App Case analysis on the head b of $b(a_1, \ldots, a_n)$:

b is an abstraction
$b(a_1, \ldots, a_n) \longrightarrow_{hnf} P(e')$ if $b(a_1, \ldots, a_n) \longrightarrow_{\beta} e \longrightarrow_{hnf} P(e')$

b is a variable or constructor
$b(a_1, \ldots, a_n) \longrightarrow_{hnf} \mathit{Hnf}(b(a_1, \ldots, a_n))$

b is a defined constant c
$c(a_1, \ldots, a_n) \longrightarrow_{hnf} \mathit{NotYet}(c(a_1, \ldots, a_n))$ if c is yet unknown
$c(a_1, \ldots, a_n) \longrightarrow_{hnf} P(e')$ if $c \longrightarrow_{\delta} e$ and $e(a_1, \ldots, a_n) \longrightarrow_{hnf} P(e')$

b is a constant defined by pattern matching
$b(a_1, \ldots, a_n) \longrightarrow_{hnf} P(e')$ if $b(a_1, \ldots, a_n) \longrightarrow_{M} e \longrightarrow_{hnf} P(e')$

4.4 Conversion

The check for conversion is done in three stages. The first stage checks syntactic equality or if either of the terms are yet unknown, the second stage assures that the type of the terms are ground and the last stage performs the reduction to head normal form and then does the (simpler) head conversion check. To compute

$$Conv(\Sigma, \Gamma, \alpha, e_1, e_2)$$

yielding a set of constraints or $Fail$, we will proceed as follows

1. if $e_1 \equiv e_2$ then $Conv(\Sigma, \Gamma, \alpha, e_1, e_2) \Rightarrow \emptyset$ (empty set of constraints)

2. if either e_1 or e_2 is a yet unknown constant, then $Conv(\Sigma, \Gamma, \alpha, e_1, e_2) \Rightarrow \{(e_1, e_2, \alpha, \Gamma)\}$

3. if $\alpha \equiv (x_1 : \alpha_1, \ldots, x_n : \alpha_n)\beta$ then
$$e_1(y_1, \ldots, y_n) = e_2(y_1, \ldots, y_n) : \beta\{x_1 := y_1, \ldots, x_n := y_n\}$$
in the context Γ extended by the new variables $y_1, \ldots, y_n$.

   This rewriting of the original conversion problem is a combination of performing 1) η-expansion, 2) α-conversion and 3) removing common abstractions of both terms, but it is done in one step. After this transformation, we know that the type must be ground, which means that when both sides are reduced to head normal form, neither term is an abstraction and in an application, the head is supplied with all its arguments (useful for pattern matching).

4. apply the reduction to head normal form to both terms, and depending on their status of the terms we have the following table

$e_2 \backslash e_1$	$Hnf(e_1)$	$Irr(e_1)$	$NotYet(e_1)$
$Hnf(e_2)$	* see below	$Fail$	ξ
$Irr(e_2)$	$Fail$	if $e_1 \equiv e_2$ then $\emptyset$ else $Fail$	ξ
$NotYet(e_2)$	ξ	ξ	ξ

where $\xi = \{(e_1, e_2, \alpha, \Gamma)\}$.

* we only have to consider two cases in the head conversion, since we know that e_1 and e_2 are on head normal form, and since the type is ground, they are not abstractions. The only possibilities are for the heads to be either variables or constructors, applied to a proper number of arguments.

$b(a_1, \ldots, a_n)$ and $b'(a'_1, \ldots, a'_n)$

if $b \equiv b'$ then

$$\frac{\begin{array}{c} Conv(\Sigma, \Gamma, \alpha, a_1, a'_1) \Rightarrow \xi_1 \\ \vdots \\ Conv(\Sigma, \Gamma, \alpha, a_n, a'_n) \Rightarrow \xi_n \end{array}}{Conv(\Sigma, \Gamma, \alpha, b(a_1, \ldots, a_n), b'(a'_1, \ldots, a'_n)) \Rightarrow \bigcup_{i=1}^{n} \xi_i}$$

else $Fail$ (for $b \not\equiv b'$).

b and b'

$$Conv(\Sigma, \Gamma, \alpha, b, b') \Rightarrow \begin{cases} \emptyset & \text{if } b \equiv b' \\ Fail & \text{otherwise} \end{cases}$$

5 Environment

The environment contains three parts, the theory declarations, the dictionary and the scratch area. The declarations in the theory and dictionary are required to correspond to true judgements, while in the scratch area valid judgements are allowed. This means that the constants in the scratch area can depend on any primitive or defined constant which is declared in the environment, but the other way around is not accepted.

The environment is extended by one declaration at the time, and any kind of definition can be added at any time, provided the new declaration is a *valid extension* of the current environment. The only operation on the theory and the dictionary is the extension of a new declaration which is explained below. The extension is monotone in the sense that if a judgement is true in an environment, it is also true in any valid extension of that environment. (Note that the monotony property does not include the scratch area definitions). The scratch area and its operations are described in the next two sections.

5.1 Extending the theory

The theory contains both declarations of canonical constants (constructors) and non-canonical constants (implicitly defined constants) which in addition to their type declaration have a set of pattern equations. Therefore the extension is either a named type declaration (canonical or non-canonical constant) or it is an extension of a pattern equation.

type declaration

A new primitive constant can be declared, if the type α is valid in the given context Γ with respect to the current environment Σ:

$$\frac{Valid(\Sigma) \qquad \Gamma \vdash_\Sigma \alpha : Type \qquad c \notin \Sigma}{Valid(\Sigma, c : \alpha \quad \Gamma)}$$

pattern equation

A pattern equation is an equality

$$c(p_1, \ldots, p_n) = e$$

where c must be known in the current environment Σ and c must be of arity n. The *pattern context* (Δ) and the type (β) of $c(p_1, \ldots, p_n)$ are computed, and e is checked to be of type β (in Δ). Further requirements are that redundant patterns only occur in positions where the term is uniquely determined by type checking, and that the new pattern is non-overlapping with respect to all the other pattern equations associated with c. The non-overlapping property as well as exhaustive pattern equations are ensured if the pattern equation generator is used to create new equations (see section 6.3).

$$\frac{Valid(\Sigma) \qquad \Gamma + \Delta \vdash_\Sigma c(p_1, \ldots, p_n) : \beta \qquad \Gamma + \Delta \vdash_\Sigma e : \beta}{Valid(\Sigma, c(p_1, \ldots, p_n) = e : \beta \quad \Gamma + \Delta)}$$

5.2 Extending the dictionary

The dictionary can be extended by abbreviations for a term, a context or a substitution, which correspond to true judgements. The formal rules are the following

$$\frac{\begin{array}{c} Valid(\Sigma) \\ \Gamma \vdash_\Sigma e : \alpha \\ c \notin \Sigma \end{array}}{Valid(\Sigma, c = e : \alpha \quad \Gamma)} \qquad \frac{\begin{array}{c} Valid(\Sigma) \\ \vdash_\Sigma \Gamma : Context \\ c \notin \Sigma \end{array}}{Valid(\Sigma, c = \Gamma)} \qquad \frac{\begin{array}{c} Valid(\Sigma) \\ \Gamma \vdash_\Sigma \gamma : \Delta \\ c \notin \Sigma \end{array}}{Valid(\Sigma, c = \gamma : \Delta \quad \Gamma)}$$

6 Scratch area

The scratch area is where proof development take place. A goal is represented as a yet unknown constant declaration, $c = ? : \alpha \quad \Gamma$, where α is the type corresponding to the proposition to be proven, Γ contains the free variables (or parameters) which are allowed to occur in the proofterm, which is yet to be filled in. c is the name of the proof, or more correctly, an abbreviation of the proofterm. When the proof is completed, i.e. the ? is replaced by a term which can be checked to be of type α (in Γ), the proof can be moved to the dictionary as an abbreviation, and later used as any other abbreviation. New goals can be added to the scratch area at any point, and it is possible to work on any goal one desires, simply be always referring to the name of the goal.

The two main purposes for the scratch area, is to achieve full flexibility of combining top-down proof derivation with bottom-up, as well as being able to backtrack and choose an alternative way of constructing the proof. Here, backtracking means more than "resume at an earlier state of the scratch area", since what we refer to as backtracking is rather deletion of a subterm of a proof, with the effect that only parts of the proof, structurally depending on the deleted term, will be deleted as well.

The price we have to pay for this flexibility, is that besides the set of *constant declarations* which are the actual goals and subgoals, we need to keep track of some additional information. Since the declarations in the theory part and dictionary are required to correspond to complete judgements, all the not completed declarations are kept in the scratch area, and therefore the set of constraints restricting the instantiation of the subgoals, is kept here as well. We also need a *dependency graph* which records all the dependencies between the constants, to prevent circular definitions of constants (recall that abbreviations are not allowed to be recursive). Since the actual order of the declarations in the scratch area is irrelevant (any constant declared in the scratch area can be used anywhere, before or after it is defined), and we sometimes need to sort the declarations in a topological order (dependency order), the dependency graph is very useful for this purpose as well.

To be able to perform backtracking properly, we need to distinguish between constants which are defined by *refinement* and constants which are instantiated by unification (required by constraints), since these requirements might not be relevant after the backtracking. Therefore we have a *backtrack area* which contains a copy of the scratch area declarations, but which constants are not instantiated by unification restrictions. The reason for choosing double representation of the same information, instead of performing the instantiations when needed, is that in an interactive system (especially with a window interface) we really want the information to be visible and updated continuously. This would require unnecessary computations to keep the declarations up to date most of the time. Instead, we recompute the scratch area from the backtrack area only after the (hopefully more seldom) operation of backtracking.

Finally, there is a list of *waiting patterns*, which are patterns generated by ALF, and which are waiting to get their left-hand side of the equation (their definition). A waiting patterns can be connected to a goal, and this provides a way of constructing the term within the scratch area, which is an advantage if the term is large and complicated.

6.1 Dependency graph

The dependency graph is a graph, recording the internal dependencies of the constants declarations in the scratch area. The graph provides answers to the following questions

- which constants the term of c depends on? (denoted $\mathit{TermDep}(c)$)
- which constants the type and context of c depends on? (denoted $\mathit{TypeDep}(c)$)
- all constants that c depends on (denoted $\mathit{Dep}(c) \equiv \mathit{TermDep}(c) \cup \mathit{TypeDep}(c)$)
- the set of constants which are yet unknown (denoted U)
- the constants which are yet unknown, that c depends on? ($\mathit{DepU}(c) \subseteq \mathit{Dep}(c)$)

The nodes of the graph corresponds to the constants in the current set of declarations, and the pointers between the nodes are either term dependent or type/context dependent. The reason for separating the two kinds of dependency is that a constant which is type/context dependent on c, is not a meaningful declaration without c, while the term dependent constant is still meaningful (with the term removed). We will mainly think of the graph in the term dependent point of view, since it captures the structure of the proofterm, and we will refer to the *subgraph of* c as being the part of the graph corresponding to c's (fully expanded) proofterm.

6.2 Backtrack area

The backtrack area contains a set of constant declarations, which as a subset contains the constants in the scratch area, and a dependency graph concerning these constants. The constants also occurring in the scratch area are called *visible*, whereas the others are *invisible*. The intuition is that each constant declaration stands for a *step* in the derivation, the definition of the constant stands for the "rule" applied in that step, and the new constants occurring in the definition correspond to other steps in the derivation. Moreover, the invisible constants stand for intermediate steps in the proof derivation, and the visible constant definitions corresponds to derivations "collapsed" to a few major steps. If we did not instantiate constants at all by unification and also expanded the invisible constants, then the scratch area and backtrack area declarations would be identical.

The correspondence between the scratch and backtrack area can be summarized by the following points:

- All visible constants in the scratch area are also in the backtrack area.
- If a constant is defined in the scratch area, but unknown in the backtrack area, then its definition is forced by unification.

6.3 Waiting patterns

A waiting pattern is a tuple $(c(p_1, \ldots, p_n), \alpha, \Delta, \Gamma)$, where Δ is the context of the pattern variables, and Γ the local context of c. Waiting patterns are created by the operations *create pattern* and *expand pattern*. As an example, consider the implicit constant declaration of $add : (x, y : N)N$, then create pattern will add the waiting pattern

$(add(x, y), N, [x, y : N])$ (a pattern with only pattern variables)

to the list of waiting patterns, and expanding that pattern with respect to the variable y would replace the same by

$(add(x, 0), N, [x : N])$ and $(add(x, succ(y')), N, [x, y' : N])$.

Note that the generation of waiting patterns by the operations *create pattern* and *expand pattern*, preserves the the property of the patterns of being exhaustive and non overlapping. Any of these new waiting patterns can again be expanded with respect to another pattern variable. The waiting pattern can be associated with a goal in the scratch area, which means that when that goal is proven, the pattern is completed and can be moved to the theory. A waiting pattern can also be completed by giving the definition directly.

7 Operations on the scratch area

The operations on the scratch area can be divided into three groups, *proof construction*, *proof completion* and *backtracking* (or proof deletion). Proofs are constructed by refining a yet unknown constant, i.e by giving a (partial) proofterm as its definition. The term can contain other constants from the scratch area, constants from the theory or dictionary or variables from the local context of the constant, besides the ordinary λ-term constructions. When a proof is completed in a satisfactory way, it can be moved from the scratch area to the dictionary, and from that point it is no longer possible to modify. On the other hand, as long as the proof is still in the scratch area (completed or not) it can be modified by these operations. There are two different ways of backtracking, by removing the definition of a constant or by removing the entire constant declaration. The former operation can be useful for retrying a (not so successful) attempt of constructing a proof, and the latter when a lemma or subgoal is declared which was found not to be useful.

7.1 Proof construction

For constructing a proof we have the following operations available; extending the scratch area with new goals or lemmas, making a global replacement of a constant with its definition (unfolding), and refining a goal. The refinement of a goal can be done top down - by refining with a constant which generates new subgoals, or bottom up - by explicitly giving the proofterm, or a combination of the two. Each refinement is checked to be *admissible*, which means that the set of constraints is not violated.

Extending the scratch area

The scratch area can be extended with a new constant declaration, a new named context or substitution, or a new waiting pattern. The constant declaration can be yet unknown (new goal) or it can be partially or totally defined. Contexts and substitutions can depend on yet unknown constants as well (and therefore give rise to constraints) but can not be altered. A new waiting pattern can be added, refined, completed, and moved to the theory without changing the rest of the scratch area. To summarize, any extension of the scratch area is allowed at any time, *provided* that the set of constraints is not violated.

Unfolding a definition

If $c = e : \alpha \quad \Gamma$ is a declaration in the scratch area, then c is replaced everywhere in the set of declarations by e, and the declaration is removed. The dependency graph is updated, which means that each incoming pointer of node c is replaced by the set of outgoing pointers and the node is removed. The set of constraints is not affected by this operation, since all definitions are always expanded as far as possible in the constraints, and the only change in the backtrack area is that c becomes invisible.

Top down and bottom up refinements

To refine a constant c with a term e, we need to type check if e is of proper type. But to be able to type check, all constants in e must be known in advance, which means that they must be declared in the environment. Therefore is the term given as a refinement, "preprocessed" before type checking. The preprocess consists of doing the following

- if the arity of e is greater than the arity of c, which means that e *needs more arguments* to be of proper type, then e is applied to a proper number of system generated *new names*
- a *new name* in the term is considered to be a new subgoal, which means that the type and context of such a subgoal is computed, and the new unknown constant declaration is added to the scratch area. The computation of type and context is done as for pattern variables.
- if the symbol _ is given instead of a new name, the system will generate a unique name. All system generated names become by default *invisible constants*, whereas user names become visible. The purpose of this setup is to name *important subgoals* and neglect (by using _) the others.
- all the transformations above are done recursively in the subterms, for partially given terms.

Admissible refinement

The algorithm for checking if the refinement of the constant c with the term e is admissible, proceeds with the following three steps, producing a refined scratch area if the refinement is admissible, and a $Fail$ otherwise.

1. Preconditions
   - $c = ? : \alpha \quad \Gamma \in \Sigma$ (c is a yet unknown constant)
   - $\mathcal{J}(\Sigma, e : \alpha \quad \Gamma) \Rightarrow \xi$ (the type checking does not fail)
   - $c \notin \mathit{DepU}(e)$ (the new definition is not circular, with respect to c)
2. Updating the scratch area
   - ? is replaced by e in the declaration of c
   - c is replaced by e everywhere in the set of constraints and ξ is added to that set.
   - the dependency graph is updated, i.e $\mathit{TermDep}(c)$ = the set of constants occurring in e.
   - ? is replaced by e in the backtrack area
   - *if* c is invisible, then c is unfolded and removed (the operation *Unfold*)
3. Refining the scratch area
   - the set of constraints is refined, which means that each constraint $(e_1, e_2, \alpha, \Gamma)$ is rechecked by
   $$\mathcal{J}(\Sigma, e_1 = e_2 : \alpha \quad \Gamma) \Rightarrow \xi$$
   and replaced by ξ. If $\mathcal{J}$ fails, the entire refinement is rejected. The reason for refining the set of constraints is that the replacement of e for c might have created a failure constraint
   - if the set of constraints contain a *simple constraint*, $(c' = e' : \beta \quad \Delta, whichisnotcircular)$, then the c' is updated to e' in the scratch area, as in (2), *except* for the replacement in the backtrack area, which remains the same. The reason to keep constraints, which are circular, is that if $x = y(x)$ is the constraint, then the equality will hold if y is instanciated to $[z]0$, for example.

These two steps are repeated until there are no simple constraints (and the refinement is *admissible*) or some refined constraint produces a $Fail$, and the refinement is *not admissible.*

7.2 Proof completion

This operation consists of two parts, to check that the construction of the constant is completed and if so, move the constant declaration to the dictionary.

A complete proof
: A precise definition of a *complete* constant (proof) is that c is complete iff $c \notin U$ and $DepU(c) = \emptyset$. A complete definition or judgement is when all constants occurring in the definition/judgement are complete. A proof can consist of several constant declarations, one main constant (corresponding to the proof), and other minor constant declarations (corresponding to lemmas). All these constants must be in the subgraph of the proof, which means that all lemmas must be used in the proof.

Moving a complete proof to the dictionary
: To move a proof c from the scratch area to the dictionary the following steps needs to be done

- check that c is complete, and compute the set of constants $\{c_1, \ldots, c_n\}$, which is used in c.
- sort $c \cup \{c_1, \ldots, c_n\}$ in their topological order (c will always be last), and add the constant declarations to the dictionary in this order, which will satisfy the condition of a valid extension.
- the declarations of c and $\{c_1, \ldots, c_n\}$ are simply erased from the scratch area (including the backtrack area). There is no problem in erasing the declarations, since the constants are complete and since they are *moved* and not removed (and can still appear in other declarations). The subgraph of c is removed from the dependency graph, since it only concerns scratch area constants.

7.3 Backtracking

Any deletion of a constant or a constant definition, are proceeded by an allowance check, before the actual deletion is performed in the backtrack area, and finally the new scratch area is computed from the diminished backtrack area. We will proceed by first describing the recomputation of a scratch area, and then the two kinds of deletion procedures are explained. Note that the constant declarations and dependency graph mentioned in these explanations, refer to the backtrack area.

Recompute the scratch area
: Recomputation of the scratch area is done after any kind of backtracking, since the old scratch area is not of any use at that point. The following steps will compute a scratch area from a backtrack area, which will have the same correspondence between them as described in section 6.2.

- Unfold the definitions of all invisible constant declarations everywhere,
- sort the remaining declarations topologically,
- extend an empty scratch area with the constant declarations in order, yielding a set of constraints,
- and finally, refine the scratch area constraints as in step (3) of the refine operation.

Remove a constant declaration

The entire constant declaration $c = e : \alpha \quad \Gamma$ is *removable* if there are no other constants depending on any of the constants $c, c_1, \ldots, c_n$, where $\{c_1, \ldots, c_n\} = Dep(c)$, except internal dependencies between these constants. The reason for this requirement is simple - if there is a declaration in which the type or context depend on, say c, then the same declaration would not make sense if c is removed. Describing that c is removable in terms of dependency graph, is to say that the subgraph of c must be *disconnected* from the rest of the graph.

Remove the definition of a constant

The definition e of a constant declaration $c = e : \alpha \quad \Gamma$ can be removed (replaced by ?) if the set of constants $\{c_1, \ldots, c_n\}(\equiv DepU(e))$, which e depends on, are all *removable*. The reason we want to remove the constant declarations of $c_1, \ldots, c_n$ is that they all correspond to steps in the derivation of c, and we want to remove the entire derivation. To be more precise, the definition of c is allowed to be removed if the subgraph of c is only connected externally (to other parts of the graph) via the node c. The actual procedure of removing the definition take place in three steps, by first replacing the definition of c with ?, then removing the constant declarations of $c_1, \ldots, c_n$ and finally recompute the scratch area from this revised backtrack area. In the dependency graph, the subgraph of c except the node c itself, is deleted.

The only reason for not allowing a removal of a constant or definition is that this operation would remove some constant which is used elsewhere. Sometimes this can be arranged by further removals of constants and/or definitions, but we have taken the stand point not to do any "clever" decisions of what to remove (besides the obvious parts) and instead disallow removals of this kind. Since anything is possible to remove, as long as it is done in the proper order, the restriction seems motivated.

8 Validity of the scratch area - future work

As already mentioned, this is an implementation of experimental nature trying out several ideas simultaneously. The scratch area involves most of these ideas, and this part needs to be described more formally, hopefully reaching at a point where we can state and prove properties such as

- if the constraints concerning a definition is satisfied, then the definition corresponds to a true judgement (this is now given as a definition)
- define the relation between scratch area and backtrack area, and show that the relation is preserved for the different operations on the scratch area

- define the partial order of "definedness" concerning the scratch area, and show that backtracking will always result in a "less defined" scratch area

9 Conclusion

Since this new implementation is still under developement, we will have to do much more experiments to be able to evaluate the new features. However, a few larger examples are carried out in ALF, such as a normalization proof for simply typed λ calculus [Coqa], another normalization proof for the typed combinator calculus [GS], as well as some properties of well-quasi ordered sets [Fri].

References

[ACN90] L. Augustsson, T. Coquand, and B. Nordström. A short description of Another Logical Framework. In *Proceedings of the First Workshop on Logical Frameworks, Antibes*, pages 39–42, 1990.

[Coqa] Catarina Coquand. A proof of normalization for simply typed lambda calculus written in alf. In *To appear in the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[Coqb] Thierry Coquand. Pattern matching with dependent types. In *To appear in the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[Fri] Daniel Fridlender. Formalizing properties of well-quasi ordered sets in alf. In *To appear in the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[GS] Veronica Gaspes and Jan M. Smith. Machine checked normalization proofs for typed combinator calculi. In *To appear in the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[Mag91] Lena Magnusson. An Implementation of Martin-Löf's Logical Framework. Licentiate Thesis, Chalmers University of Technology and University of Göteborg, Sweden, June 1991.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

An implementation of Constructive Set Theory, in the Lego system

Nax Paul Mendler
Computer Science Department
University of Manchester
Manchester M13 9PL

August,1992

Remark, by Peter Aczel: *This note was written by Nax Mendler, just before he left the LF project in November 1991. My talk at the workshop was a report on his work.*

This note contains comments to accompany a file library I wrote for Randy Pollack's Lego system.

In the paper [1] Peter Aczel described a constructive set theory (CST) and gave an interpretation of it into Martin-lof type theory. I took this paper and implemented this interpretation in the Extended Calculus of Constructions using Lego. The best and perhaps only way to study this Lego file is to read it or enter it into the system as you follow along in the original paper, and I've tried to have the Lego file match it as closely as possible.

In Lego I used the Extended Calculus of Constructions theory [2] and extensively used the new facility for adding user-defined combinators and reductions. The most important example of this is the type called V, used to interpret the universe of sets: it is the well-founded type $(Wx : \mathit{Type}_0)x$, and its induction combinator was used in defining a bisimulation relation which models extensional equality on sets.

The only subtle technical point of the Lego implementation was that, while the original paper assumed the existence of extensional identity types in the type theory, one can rework a few arguments and use definitional equality instead. This is not too surprising: I think it soon becomes clear whether one can or can't eliminate the use of extensional identity types in a given piece of type theory, and in this case I was lucky.

So, in this Lego library, the propositions-as-types interpretations or translations of the axioms of CST are all "proven." The intention is that one can then go on to do some set theory in Lego and also be able to execute the constructive content of the proofs. One example that we thought might be of a reasonable size is to define and prove some basic facts about the hereditarily finite sets.

Just a few sentences on what this experiment has taught me about the Lego system. I've used Lego before, although this was my first long example using the combinator definition facility. As I've come to expect from the system, what it is meant to do it does with style. I already knew the basic definition mechanism and type synthesis would prove useful, and I was pleased with the new user-defined combinator facility. I was happy with how fast the system ran, too. But as for the question of why do any formal theorem proving on computer, I still can not be positive — my example should only be viewed as evidence that it is possible to sit down and enter these sorts of formal proofs, not as an endorsement of the activity.

References

[1] Peter Aczel. The type theoretic interpretation of constructive set theory: Choice principles. In A.S. Troelstra and D. van Dalen, editors, *The LEJ Brouwer Centenary Symposium*, pages 1–40. North-Holland, 1982.

[2] Z.Luo. A higher-order calculus and theory abstraction. *Information and Control*, 90(1):107-137, January 1991.

lambda mu-calculus: an algorithmic interpretation of classical natural deduction

Michel Parigot, Paris 7

Abstract

We present a way of extending the paradigm "proofs as programs" to classical proofs. The system used is a natural deduction system with multiple conclusions. It has two kinds of reduction rules: logical and structural ones (logical ones correspond exactly to those of intuitionistic natural deduction).

The computation mechanism can be described as a pure calculus, called lambda mu-calculus, which extends in a simple manner lambda-calculus (mu looks like a lambda having a potentially infinite number of arguments). Pure lambda mu-calculus satisfies the Church-Rosser property. Typed lambda mu -calculus satisfies also the type preservation and strong normalisation properties.

lambda mu-calculus has a simple abstract environment based machine: the new instructions (in addition of those of lambda-calculus) are "save the stack" and "restore the stack".

Teaching Theory of Programming Languages Using a Logical Framework: an Experience Report

Frank Pfenning, Carnegie Mellon

Abstract

During the Spring'92 semester I taught a graduate course "Computation and Deduction". This course explored the theory of programming languages using systems of natural deduction. Such systems were used to specify, implement, and verify properties of functional, imperative, and logic programming languages. The deductive approach to the specification of programming languages has become standard practice, and one of the goals of this course was to provide a good working knowledge of how to engineer and prove properties of language descriptions in this style. Throughout the course we used Elf as a meta-language. Elf is a logic programming language based on the LF Logical Framework and was introduced by means of a few extended examples. An implementation of Elf and all the examples were available on-line for experimentation. Overall there were 29 lectures of 1.5 hours each, of which 12 were given by students presenting projects.

The rigorous use of Elf as a metalanguage was seen as a very positive factor. Otherwise dry exercises became programming problems, and the students learned how to employ Elf effectively to implement languages and their meta-theory. Type reconstruction and a few other features of the Elf implementation turned out to be usable and valuable. It was generally felt that the expressive power of the LF logical framework was adequate, although a few extensions emerged as desirable in practice. These were a simple definitional equality, a limited form of subtypes and overloading, and a module system.

In my talk, I will outline the approach and content of the course. Furthermore, I will sketch some initial ideas on how to enhance the expressive power of LF in directions motivated by the course experience.

Typechecking in Pure Type Systems

Randy Pollack
LFCS, University of Edinburgh

July 1992

Retraction

At Baastad I claimed a proof of the Expansion Postponment property for functional PTS. Eric Poll of Eindhoven pointed out an error in my claimed proof. As far as I know this problem is still open.

1 Introduction

This work is motivated by two related problems. The first is to find reasonable algorithms for typechecking Pure Type Systems [Bar91] (PTS); the second is a technical question about PTS, the so called Expansion Postponment property (EP), which is tantalizingly simple but remains unsolved.

There are several implementations of formal systems that are either PTS or closely related to PTS. For example, LEGO [LP92] implements the Pure Calculus of Constructions [CH88] (PCC), the Extended Calculus of Constructions [Luo90] and the Edinburgh Logical Framework [HHP87]. ELF [Pfe89] implements LF; CONSTRUCTOR [Hel91] implements arbitrary PTS with finite set of sorts. Are these implementations actually correct? It is not difficult to find a natural efficient algorithm that is provably sound (Section 3), but completeness is more difficult. In fact Jutting has shown typechecking is decidable for all normalizing PTS with a finite set of sorts [vBJ92], but his algorithm, which computes the normal forms of types, is not suitable for practical use.

1.1 The Decision Problems

We consider two problems about PTS. The Type Checking Problem (TCP) is to decide, given Γ, M, and A, whether or not $\Gamma \vdash M : A$ is derivable. In implementations we want to enter a term and have the program compute its type in the current context. For PTS with types unique up to conversion, the functional PTS, this Type Synthesis Problem (TSP) is clear: given Γ and M, compute a term A such that $\Gamma \vdash M : A$, or return failure if no such A exists. In general, however, PTS do not have unique types, and an exact statement of TSP involves a notion of type scheme which will be postponed to Section 2.3.

Let s range over $\{\mathsf{Prop},\mathsf{Type}\}$, the *sorts* of PCC.

AX $\qquad \bullet \vdash \mathsf{Prop} : \mathsf{Type}$

START $\qquad \dfrac{\Gamma \vdash A : s}{\Gamma[x{:}A] \vdash x : A} \qquad x\ \textit{fresh}$

WEAK $\qquad \dfrac{\Gamma \vdash \alpha : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash \alpha : C} \qquad x\ \textit{fresh}$

PI $\qquad \dfrac{\Gamma[x{:}A] \vdash B : s}{\Gamma \vdash \{x{:}A\}B : s}$

LDA $\qquad \dfrac{\Gamma[x{:}A] \vdash M : B}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B} \qquad B \neq \mathsf{Type}$

APP $\qquad \dfrac{\Gamma \vdash M : \{x{:}A\}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : [N/x]B}$

CONV $\qquad \dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \simeq B}{\Gamma \vdash M : B}$

Table 2: The Pure Calculus of Constructions (PCC)

1.2 An Example: Typechecking Pure Constructions

Without being too precise for the moment Table 2 presents typing rules for the Pure Calculus of Constructions (PCC). Consider the type synthesis problem for PCC: given Γ and M, compute a term A such that $\Gamma \vdash M : A$, or return failure if no such A exists. The natural approach is to look for a derivation of $\Gamma \vdash M : \_$ guided by the shape of the subject, Γ and M. The only flaw in this plan is the rule CONV, which may be used at any point in a derivation without changing the shape of the subject. Putting it the other way around, you cannot tell when to use CONV in a derivation by looking at the shape of the subject. It is clear we must control the CONV rule. In my first typechecker for PCC, coded in 1986, my approach was to compute the normal forms of types. This eliminates the need for CONV, and, since PCC is normalizing, leads to a sound and complete algorithm if done correctly. (For example, you must check that a term is well-typed *before* trying to normalize it!) This approach is terribly slow, and clearly could not be used to check large bodies of mathematics.

In 1987, Gérard Huet showed me a collection of techniques for efficiently typechecking PCC that he collected into an abstract machine called the Constructive Engine [Hue87]. The most important aspect of the Constructive Engine is shown abstractly in Table 3. The correctness lemma for this system is

Lemma 1.1 *(1) If $\Gamma \vdash_{\mathrm{ce}} M : A$ then $\Gamma \vdash M : A$.*
(2) If $\Gamma \vdash M : A$ then there exists A' such that $A \simeq A'$ and $\Gamma \vdash_{\mathrm{ce}} M : A'$

Table 2, because of its non-determinstic use of CONV, allows many derivations over a given subject, even deriving structurally different types for a single subject. (It is a property of PCC that all those types are convertible.) In contrast, the system of Table 3 has no CONV rule and allows at most one derivation over a given Γ and M, and that derivation uniquely determines

Notation: we write $\Gamma \vdash_{\mathrm{ce}} M :\geq A$ for $\Gamma \vdash_{\mathrm{ce}} M : A'$ and $A' \geq A$, where $\geq$ is β-reduction and $\overset{wh}{\geq}$ is β-weak-head-reduction.

CE-AX	$\bullet \vdash_{\mathrm{ce}} \mathsf{Prop} : \mathsf{Type}$	
CE-START	$\dfrac{\Gamma \vdash_{\mathrm{ce}} A :\geq s}{\Gamma[x{:}A] \vdash_{\mathrm{ce}} x : A}$	*x fresh*
CE-WEAK	$\dfrac{\Gamma \vdash_{\mathrm{ce}} \alpha : C \qquad \Gamma \vdash_{\mathrm{ce}} A :\geq s}{\Gamma[x{:}A] \vdash_{\mathrm{ce}} \alpha : C}$	*x fresh*
CE-PI	$\dfrac{\Gamma[x{:}A] \vdash_{\mathrm{ce}} B :\geq s}{\Gamma \vdash_{\mathrm{ce}} \{x{:}A\}B : s}$	
CE-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\mathrm{ce}} M : B}{\Gamma \vdash_{\mathrm{ce}} [x{:}A]M : \{x{:}A\}B}$	$B \neq \mathsf{Type}$
CE-APP	$\dfrac{\Gamma \vdash_{\mathrm{ce}} M :\overset{wh}{\geq} \{x{:}A\}B \qquad \Gamma \vdash_{\mathrm{ce}} N : A' \qquad A \simeq A'}{\Gamma \vdash_{\mathrm{ce}} M\,N : [N/x]B}$	

Table 3: A Preliminary Constructive Engine: syntax-directed PCC

the type it derives for M in Γ. Consequently Table 2 cannot derive all the structurally different types for M in Γ, but only one representative of the conversion class.

Notice that for correctness we could allow arbitrary reduction sequences where we have written weak-head reduction in CE-APP, but the restriction to deterministic weak-head reduction is necessary to make this rule syntax directed; otherwise we wouldn't know when to stop reducing in the left premiss, and A and B wouldn't be uniquely determined.

We can think of Lemma 1.1 as saying all uses of CONV for expanding a type can be permuted downwards in a derivation through all premisses of all rules, and all uses of CONV for reducing a type can be permuted downwards in a derivation through the left premiss of WEAK and the premiss of LDA. The fact that Table 3 is syntax directed is extremely delicate, although we didn't know this in 1987 because the proof of Lemma 1.1 is quite straightfoward. In PTS it is still unproven that type expansion can always be deferred to the end of derivations, and we know by counterexample (Example 3.2) that type reduction cannot always be permuted through the premiss of LDA.

Table 3 can already be thought of as a sound and complete type synthesis algorithm for PCC, but there is one other significant optimization in the Constructive Engine that is worth mentioning, even though it is not delicate, and gives no problems in the case of PTS. In the rules CE-WEAK and CE-APP Γ occurs in two premisses, and its validity must be checked in the subderivations above both premisses. It is much more efficient to assume we start with a valid context, and only check that it remains valid whenever we extend it. This is also more in keeping with the use of actual implementations, where we want to work in a "current context" of mathematical assumptions (and also of definitions and proved theorems, but I do not address definitions in this note). This is shown abstractly in Table 4. The new premisses in CEO-PI and CEO-LDA are to check that the extended contexts in the second premisses are valid, assuming that Γ is valid. The correctness lemma for this system is

Notation: we write $\Gamma \vdash_{\text{ceo}} M :\geq A$ for $\Gamma \vdash_{\text{ceo}} M : A'$ and $A' \geq A$.

CEO-VALID	$\bullet\ \mathsf{Valid}$	$\dfrac{\Gamma\ \mathsf{Valid} \qquad \Gamma \vdash_{\text{ceo}} A :\geq s}{\Gamma[x{:}A]\ \mathsf{Valid}}$	x *fresh*
CEO-PROP	$\Gamma \vdash_{\text{ceo}} \mathsf{Prop} : \mathsf{Type}$		
CEO-VAR	$\Gamma[x{:}A]\Delta \vdash_{\text{ceo}} x : A$		
CEO-PI	$\dfrac{\Gamma \vdash_{\text{ceo}} A :\geq s' \qquad \Gamma[x{:}A] \vdash_{\text{ceo}} B :\geq s}{\Gamma \vdash_{\text{ceo}} \{x{:}A\}B : s}$		
CEO-LDA	$\dfrac{\Gamma \vdash_{\text{ceo}} A :\geq s \qquad \Gamma[x{:}A] \vdash_{\text{ceo}} M : B}{\Gamma \vdash_{\text{ceo}} [x{:}A]M : \{x{:}A\}B}$		$B \neq \mathsf{Type}$
CEO-APP	$\dfrac{\Gamma \vdash_{\text{ceo}} M :\overset{wh}{\geq} \{x{:}A\}B \qquad \Gamma \vdash_{\text{ceo}} N : A' \qquad A \simeq A'}{\Gamma \vdash_{\text{ceo}} M\,N : [N/x]B}$		

Table 4: An Optimised Constructive Engine

Lemma 1.2 *(1) If $\Gamma \vdash_{\text{ceo}} M : A$ and $\Gamma\ \mathsf{Valid}$ then $\Gamma \vdash M : A$.*
(2) If $\Gamma \vdash M : A$ then there exists A' such that $A \simeq A'$ and $\Gamma \vdash_{\text{ceo}} M : A'$

Setting aside the question of how to test conversion, we consider Table 4 to be a good algorithm for typechecking PCC.

1.3 The remainder of this note

For arbitrary PTS there are several new difficulties. The most troublesome is that we don't know how to eliminate the conversion rule in favor of a syntax-directed system for arbitrary PTS. The difficulty in following the Constructive Engine approach is discussed in Section 3. In Section 4 we give a syntax-directed presentation of a restricted class of PTS that includes PCC, and the Edinburgh Logical Framework (LF). (Thus, we finally have a proof of an efficient and natural algorithm for typechecking LF.) It may be of interest that the correctness of the syntax-directed system for this class of PTS has been machine checked in LEGO [LP92], except for a few facts about reduction and conversion. This class also has the Expansion Postponment property.

In PCC there are only two sorts, Prop and Type, and PCC has types unique up to conversion. In general a PTS may have infinitely many sorts and fail to have unique types. Even if we have a syntax-directed presentation for a class of PTS, where all derivations over a given subject have the same shape, undecidable and non-deterministic side conditions may complicate an algorithmic interpretation. Such issues are well understood, and I show how to handle them in Section 4.4.

2 Pure Type Systems

In order to fix notation we define PTS and state some well known properties. A PTS, $\mathcal{P}$, is a 4-tuple $(\mathsf{Cnst}, \mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule})$ where

- Cnst, a set of *constants* (ranged over by c)
- Sort $\subseteq$ Cnst, a set of *sorts*, (ranged over by s)
- Ax $\subseteq$ Cnst $\times$ Sort, a set of *axioms* of the form Ax$(c{:}s)$
- Rule $\subseteq$ Sort $\times$ Sort $\times$ Sort, a set of *rules* of the form Rule(s_1, s_2, s_3)

Syntax Let x range over Var, an infinite set of variables disjoint from Cnst. The raw syntax of *terms*, *contexts* and *judgements* of PTS $\mathcal{P} = (\mathsf{Cnst}, \mathsf{Sort}, \mathsf{Ax}, \mathsf{Rule})$ is given by:

atomic terms	α	$::=$	x	*variable*
		$\mid$	c	*constant*
terms	M	$::=$	α	*atomic*
		$\mid$	$[x{:}M]M$	*lambda*
		$\mid$	$\{x{:}M\}M$	*Pi*
		$\mid$	$M\,M$	*application*
contexts	Γ	$::=$	$\bullet$	*empty*
		$\mid$	$\Gamma[x{:}M]$	
judgement	J	$::=$	$\Gamma \vdash M : M$	

M, N, A, B, C, D, E, a, b range over terms; Γ, Δ over contexts.

$\mathsf{FV}(M)$ denotes the free variables of M, defined as usual. If $\Gamma = [x_1{:}A_1]\ldots[x_n{:}A_n]$ then $\mathsf{V}(\Gamma)$ is defined as the set $\{x_1, \ldots, x_n\}$, and $\mathsf{FV}(\Gamma)$ as the set $\bigcup\{\mathsf{FV}(A_1), \ldots, \mathsf{FV}(A_n)\}$. If $\Gamma = \Gamma_1[x{:}A]\Gamma_2$ we say $x{:}A \in \Gamma$. $\Gamma \subseteq \Delta$ is defined to mean $x{:}A \in \Gamma$ implies $x{:}A \in \Delta$.

Reduction Substitution, β-reduction, and β-conversion are defined as usual for such languages. Write [N/x]B for "capture-avoiding substitution of N for free ocurrances of x in B", $A \geq B$ for "A reduces to B", $A \overset{wh}{\geq} B$ for "A weak-head reduces to B" and $A \simeq B$ for "A converts to B". β-reduction has the Church-Rosser (CR) property. $\mathsf{nf}(A, B)$ is the relation "B is the normal form of A".

Typing The judgements of $\mathcal{P}$ are those derivable from the axioms and inference rules of Table 5. As usual we abuse notation by writing $\Gamma \vdash M : A$ to mean the judgement is derivable. We say Γ Valid iff there exist M and A with $\Gamma \vdash M : A$.

The cognoscenti will notice that the rule WEAK in Table 5 is not the usual one. I have restricted weakening to atomic subjects for the purpose of discussing typechecking, since this presentation is more syntax-directed. It is straightfoward to see that the full weakening rule is admissible in this system. In fact this presentation gives a better development of the basic metatheory because no case of the Generation Lemma except START depends on the Thinning Lemma. The basic metatheory of this presentation of PTS has been formalized in LEGO

2.1 Basic Theorems

The basic meta-theoretic properties of PTS are presented in [Bar92, Ber90, GN91, vBJ92]. The ones we will explicitly use are summarized here

Lemma 2.1 (Thinning Lemma) *If $\Gamma \vdash M : A$, $\Gamma \subseteq \Delta$, and Δ Valid, then $\Delta \vdash M : A$.*

AX	$\bullet \vdash c : s$	$\mathsf{Ax}(c{:}s)$
START	$\dfrac{\Gamma \vdash A : s}{\Gamma[x{:}A] \vdash x : A}$	x *fresh*
WEAK	$\dfrac{\Gamma \vdash \alpha : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash \alpha : C}$	x *fresh*
PI	$\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma[x{:}A] \vdash B : s_2}{\Gamma \vdash \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
LDA	$\dfrac{\Gamma[x{:}A] \vdash M : B \qquad \Gamma \vdash \{x{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$	
APP	$\dfrac{\Gamma \vdash M : \{x{:}A\}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : [N/x]B}$	
CONV	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \simeq B}{\Gamma \vdash M : B}$	

Table 5: The Typing Judgement of a PTS

Lemma 2.2 (Generation Lemma) *(1) If $\Gamma \vdash c : A$ then there exists s such that $\mathsf{Ax}(c{:}s)$ and $A \simeq s$.*
(2) If $\Gamma \vdash x : A$ then there exists A' such that $x{:}A' \in \Gamma$ and $A \simeq A'$.
(3) If $\Gamma \vdash \{x{:}B\}D : A$ then there exist s_1, s_2, s_3 such that $\mathsf{Rule}(s_1, s_2, s_3)$, $\Gamma \vdash B : s_1$, $\Gamma[x{:}B] \vdash D : s_2$, and $A \simeq s_3$.
(4) If $\Gamma \vdash [x{:}B]d : A$ then there exist s and D such that $\Gamma[x{:}B] \vdash d : D$, $\Gamma \vdash \{x{:}B\}D : s$, and $A \simeq \{x{:}B\}D$.
(5) If $\Gamma \vdash b\,d : A$ then there exist B, D such that $\Gamma \vdash b : \{x{:}D\}B$, $\Gamma \vdash d : D$, and $A \simeq [d/x]B$.

Lemma 2.3 (Correctness of Types) *If $\Gamma \vdash b : B$ then there exists s such that $B \simeq s$ or $\Gamma \vdash B : s$.*

Lemma 2.4 (Subject and Predicate Reduction) *If $\Gamma \vdash b : B$, $\Gamma \geq \Gamma'$, $b \geq b'$, and $B \geq B'$, then $\Gamma' \vdash b' : B'$.*

A PTS is called *functional* if $\mathsf{Ax}(c{:}s)$ and $\mathsf{Ax}(c{:}s')$ imply $s = s'$, and $\mathsf{Rule}(s_1, s_2, s_3)$ and $\mathsf{Rule}(s_1, s_2, s_3')$ imply $s_3 = s_3'$. Functional PTS have types unique up to conversion.

2.2 A First Look at Decidability

We cannot expect to decide typechecking for a PTS whose axioms or rules are not decidable (although none of the basic metatheory mentioned so far uses such an assumption), so from now on assume they are decidable.

At first, blinded by the beauty of the Constructive Engine, I conjectured that all normalizing PTS have decidable TCP. This belief is sadly mistaken:

Example 2.1 *Let $\mathsf{T}(i,n)$ mean "the i^{th} Turing machine, when started with i on it's input tape, halts in exactly n steps". For the PTS*

undecidable	Cnst	*natural numbers*
	Sort	*natural numbers*
	Ax	$\{ (i{:}n) \mid \mathsf{T}(i,n) \}$
	Rule	$\emptyset$

we have, for any i, $[x{:}i] \vdash x : i$ iff there exists n with $\mathsf{Ax}(i{:}n)$ iff the i^{th} Turing machine halts on input i. This PTS is functional and strongly normalizing (there are no well typed redices), but has undecidable TCP.

2.3 Most General Type Schemes

Is there hope to find systems of syntax-directed derivations for non-functional PTS? Suppose there are judgements of the shape $\Gamma \vdash b : \{x_1{:}B_1\}s_1$ and $\Gamma \vdash b : \{x_1{:}C_1\}\{x_2{:}C_2\}s_2$ where the types are in normal form. We wouldn't expect a derivation determined by the shape of Γ and b to construct both of these types since they have different shapes. In fact this problem doesn't arise. I believe the following property was first observed by Luo for ECC [Luo90], and independently discovered by Jutting for PTS [vBJ92].

Lemma 2.5 *If $\Gamma \vdash a : A$ and $\Gamma \vdash a : B$ then either (i) $A \simeq B$, or (ii) there exist $s_A, s_B, n \geq 0$, $C_1, \ldots, C_n$ such that $A \geq \{x_1{:}C_1\} \ldots \{x_n{:}C_n\}s_A$ and $B \geq \{x_1{:}C_1\} \ldots \{x_n{:}C_n\}s_B$*

As a corollary we have that if $\Gamma \vdash a : A$ and $\Gamma \vdash a : B$ then there is an A', obtained by replacing some of the sort occurrences in A by other sorts, such that $B \simeq A'$. We introduce some machinery to talk about this idea,

Sort Variables and Constraints Let Σ be a new class of *sort variables* disjoint from Cnst and Var, ranged over by σ. *Schematic terms*, ranged over by X, Y, Z, U, are terms that may contain sort variables as well as sorts

$$\begin{array}{lcl} \alpha & ::= & x \mid c \mid \sigma \\ X & ::= & \alpha \mid [x{:}X]X \mid \{x{:}X\}X \mid X\,X \end{array}$$

$\mathsf{SV}(X)$ is the set of sort variables occurring in X.

A *sort asignment*, ranged over by φ, is a partial function assigning sorts to a finite set of sort variables. $\mathsf{Dom}(\varphi)$ is the domain of φ as a function.

Reduction, conversion, and sort assignments are extended to schematic terms in the obvious way. We call φX an *instance* of X. Notice that φ respects the structure of terms, and respects redices, so can be thought of as a bijection between the reduction sequences from X and the reduction sequences from φX. In particular X has a normal form exactly when φX does.

Most General Type Schemes By Lemma 2.5, if $\Gamma \vdash M : A$ then there is an X such that every type for M in Γ converts with an instance of X. The problem is to find just those instances of X that actually are correct types for M in Γ.

A *constraint* is any formula whose only free variables are sort variables. We have in mind formulas of the form $\mathsf{Ax}(c{:}\sigma)$, $\mathsf{Rule}(\sigma_1, \sigma_2, \sigma_3)$, and $\sigma_1 = \sigma_2$. $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$, range over finite sets of constraints. $\mathsf{SV}(\mathcal{C})$ is the set of sort variables occurring in $\mathcal{C}$. A sort assignment *satisfies* a constraint set, written $\varphi \models \mathcal{C}$, iff $\mathsf{SV}(\mathcal{C}) \subseteq \mathsf{Dom}(\varphi)$ and each of the propositions in $\varphi\mathcal{C}$ is true. A constraint set is *satisfiable*, or *consistent* if there is some sort assignment satisfying it.

Assume we have specified a class of constraints such that the consistency of every set of such constraints is decidable. For example, for a PTS with finite Sort, the class including all formulas of the form $\mathsf{Ax}(c{:}\sigma)$, $\mathsf{Rule}(\sigma_1, \sigma_2, \sigma_3)$, and $\sigma_1 = \sigma_2$ is such a class (remember, we are assuming Ax and Rule to be decidable). Let $\mathcal{Z}$ be such a class, although we will usually keep this class implicit.

A pair $X, \mathcal{C}$ (where $\mathcal{C} \subseteq \mathcal{Z}$) is a *most general type scheme* (mgts) for Γ, M (with respect to $\mathcal{Z}$) iff (i) whenever $\varphi \models \mathcal{C}$ then $\Gamma \vdash M : \varphi X$, and (ii) whenever $\Gamma \vdash M : A$ there is a $\varphi \models \mathcal{C}$ such that $A \simeq \varphi X$.

The *Type Synthesis Problem* (TSP) for a PTS is, to find a class, $\mathcal{Z}$, and an algorithm that, given Γ and M, computes a most general type scheme, $X, \mathcal{C}$, for Γ, M with respect to $\mathcal{Z}$.

We will now show that, for a given PTS, if TSP is solvable then TCP is also solvable.

Schematic Conversion In order to compute the constraints under which two schematic terms are convertible we need a sort-variables unification algorithm:

$$\sigma_1 \overset{\{\sigma_1=\sigma_2\}}{=} \sigma_2 \qquad\qquad c_1 \overset{\emptyset}{=} c_2 \qquad\qquad x \overset{\emptyset}{=} x$$

$$\frac{X_1 \overset{\mathcal{C}}{=} Y_1 \qquad X_2 \overset{\mathcal{D}}{=} Y_2}{\{x{:}X_1\}X_2 \overset{\mathcal{C}\cup\mathcal{D}}{=} \{x{:}Y_1\}Y_2} \qquad \frac{X_1 \overset{\mathcal{C}}{=} Y_1 \qquad X_2 \overset{\mathcal{D}}{=} Y_2}{[x{:}X_1]X_2 \overset{\mathcal{C}\cup\mathcal{D}}{=} [x{:}Y_1]Y_2} \qquad \frac{X_1 \overset{\mathcal{C}}{=} Y_1 \qquad X_2 \overset{\mathcal{D}}{=} Y_2}{X_1\, X_2 \overset{\mathcal{C}\cup\mathcal{D}}{=} Y_1\, Y_2}$$

Now we have the schematic conversion algorithm:

$$\frac{\mathsf{nf}(X, X') \qquad \mathsf{nf}(Y, Y') \qquad X' \overset{\mathcal{C}}{=} Y'}{X \overset{\mathcal{C}}{\simeq} Y}$$

Lemma 2.6 *For normalizing X and Y (1) if $X \overset{\mathcal{C}}{\simeq} Y$ and $\varphi \models \mathcal{C}$ then $\varphi X \simeq \varphi Y$,*
(2) if $\varphi X \simeq \varphi Y$ then there exists $\mathcal{C}$ such that $X \overset{\mathcal{C}}{\simeq} Y$ and $\varphi \models \mathcal{C}$,
(3) it is decideable whether or not there exists a $\mathcal{C}$ such that $X \overset{\mathcal{C}}{\simeq} Y$.

This algorithm is not very efficient, and in practice we try to minimize the reduction involved in checking that two terms convert.

TCP Now assume that for a given normalizing PTS we have a solution to TSP, and we want to use that to solve TCP: we are given Γ, M, A, and want to decide whether or not $\Gamma \vdash M : A$. Let $X, \mathcal{C}$ be an mgts for Γ, A and $Y, \mathcal{D}$ be an mgts for Γ, M. If $\mathcal{C}$ is not consistent, then A is not a type in Γ; if $\mathcal{D}$ is not consistent, then M has no type in Γ. If both $\mathcal{C}$ and $\mathcal{D}$ are consistent then A and X are normalizing, so use the schematic conversion algorithm to compute $X \overset{\mathcal{E}}{\simeq} A$. If $\mathcal{C} \cup \mathcal{D} \cup \mathcal{E}$ is consistent then $\Gamma \vdash M : A$ is derivable, otherwise not. (Some of the algorithms may return explicit failure which I'm reading as returning an inconsistent constraint set.)

3 Expansion Postponment and Syntax-Directed derivations

Our first goal for finding a Type Synthesis Algorithm for PTS is to find a syntax-directed derivation system equivalent to Table 5 in some way analogous to the Constructive Engine.

ER-AX	$\bullet \vdash_{\mathrm{er}} c : s$	$\mathsf{Ax}(c{:}s)$
ER-START	$\dfrac{\Gamma \vdash_{\mathrm{er}} A : s}{\Gamma[x{:}A] \vdash_{\mathrm{er}} x : A}$	*x fresh*
ER-WEAK	$\dfrac{\Gamma \vdash_{\mathrm{er}} \alpha : C \qquad \Gamma \vdash_{\mathrm{er}} A : s}{\Gamma[x{:}A] \vdash_{\mathrm{er}} \alpha : C}$	*x fresh*
ER-PI	$\dfrac{\Gamma \vdash_{\mathrm{er}} A : s_1 \qquad \Gamma[x{:}A] \vdash_{\mathrm{er}} B : s_2}{\Gamma \vdash_{\mathrm{er}} \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
ER-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\mathrm{er}} M : B \qquad \Gamma \vdash_{\mathrm{er}} \{x{:}A\}B : s}{\Gamma \vdash_{\mathrm{er}} [x{:}A]M : \{x{:}A\}B}$	
ER-APP	$\dfrac{\Gamma \vdash_{\mathrm{er}} M : \{x{:}A\}B \qquad \Gamma \vdash_{\mathrm{er}} N : A}{\Gamma \vdash_{\mathrm{er}} M\,N : [N/x]B}$	
ER-RED	$\dfrac{\Gamma \vdash_{\mathrm{er}} M : A \qquad A \geq B}{\Gamma \vdash_{\mathrm{er}} M : B}$	
ER-EXP	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad B \geq A}{\Gamma \vdash M : B}$	

Table 6: The expansion/reduction type system (ERTS)

The only troublesome rule is CONV. The natural generalization of the Constructive Engine, Table 9, does not work, and to clarify the situation we consider some intermediate systems first.

In informally describing the Constructive Engine I suggested we view the conversion rule as two rules, one for expansion and one for reduction. The system ERTS of Table 6 makes this precise.

Lemma 3.1 $\Gamma \vdash M : A$ *iff* $\Gamma \vdash_{\mathrm{er}} M : A$.

Proof $\Rightarrow$ By induction on the derivation of $\Gamma \vdash M : A$. Assume the derivation ends with

CONV $$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \simeq B}{\Gamma \vdash M : B}$$

By IH $\Gamma \vdash_{\mathrm{er}} M : A$ and $\Gamma \vdash_{\mathrm{er}} B : s$. By Church-Rosser B and A have a common reduct, B', so $\Gamma \vdash_{\mathrm{er}} M : B'$ by RED and $\Gamma \vdash_{\mathrm{er}} M : B$ by EXP.

$\Leftarrow$ By induction on the derivation of $\Gamma \vdash_{\mathrm{er}} M : A$. If the derivation ends with ER-RED, use Predicate Reduction of PTS; if it ends with ER-EXP, use CONV. ■

Expansion Postponement Continuing by analogy with the Constructive engine, we try to prove that all uses of ER-EXP can be deferred until the end of a derivation, or equivalently that ER-EXP can be permuted downwards through all premisses of all rules. Consider the system

R-AX	$\bullet \vdash_{\mathrm{r}} c : s$	$\mathsf{Ax}(c{:}s)$
R-START	$\dfrac{\Gamma \vdash_{\mathrm{r}} A : s}{\Gamma[x{:}A] \vdash_{\mathrm{r}} x : A}$	x *fresh*
R-WEAK	$\dfrac{\Gamma \vdash_{\mathrm{r}} \alpha : C \qquad \Gamma \vdash_{\mathrm{r}} A : s}{\Gamma[x{:}A] \vdash_{\mathrm{r}} \alpha : C}$	x *fresh*
R-PI	$\dfrac{\Gamma \vdash_{\mathrm{r}} A : s_1 \qquad \Gamma[x{:}A] \vdash_{\mathrm{r}} B : s_2}{\Gamma \vdash_{\mathrm{r}} \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
R-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\mathrm{r}} M : B \qquad \Gamma \vdash_{\mathrm{r}} \{x{:}A\}B : s}{\Gamma \vdash_{\mathrm{r}} [x{:}A]M : \{x{:}A\}B}$	
R-APP	$\dfrac{\Gamma \vdash_{\mathrm{r}} M : \{x{:}A\}B \qquad \Gamma \vdash_{\mathrm{r}} N : A}{\Gamma \vdash_{\mathrm{r}} M\,N : [N/x]B}$	
R-RED	$\dfrac{\Gamma \vdash_{\mathrm{r}} M : A \qquad A \geq B}{\Gamma \vdash_{\mathrm{r}} M : B}$	

Table 7: The reduction type system (RTS)

RTS of Table 7. It is trivial that RTS is sound for PTS, i.e. that $\Gamma \vdash_{\mathrm{r}} M : A$ implies $\Gamma \vdash M : A$. However the *expansion postponment* conjecture, that RTS is complete for PTS

Conjecture 3.1 (Expansion Postponment) *If* $\Gamma \vdash M : A$ *then there exists* A' *such that* $A \geq A'$ *and* $\Gamma \vdash_{\mathrm{r}} M : A'$

has resisted all attempts to date for any interesting class of PTS. (In Section 4 I show that a class called semi-full, generalizing the full PTS, has this property.)

If we could prove Conjecture 3.1, would we have a syntax directed presentation of PTS? Asking which premisses R-RED can be permuted downwards through, we get the system NSDTS of Table 8, which is equivalent to RTS.

Lemma 3.2 *(1) If* $\Gamma \vdash_{\mathrm{nsd}} M : A$ *then* $\Gamma \vdash_{\mathrm{r}} M : A$
(2) If $\Gamma \vdash_{\mathrm{r}} M : A$ *then there exists* A' *such that* $\Gamma \vdash_{\mathrm{nsd}} M : A'$ *and* $A' \geq A$

This is straightfoward to prove.

In the Constructive Engine, reduction could be deferred through the premiss of LDA. In NSDTS, reduction gets "stuck" at the left premiss of NSD-LDA, and for this reason NSDTS is not syntax-directed: we cannot tell from the shape of $[a{:}A]M$ how much reduction to do to find B. Can we hope to remove this deficiency? Consider system SDTS of Table 9, which is identical to NSDTS except it doesn't allow reduction after the left premiss of the lambda rule. Clearly $\Gamma \vdash_{\mathrm{sd}} M : A$ implies $\Gamma \vdash_{\mathrm{nsd}} M : A$, so we have, somewhat inexactly,

$$\mathrm{SDTS} \Rightarrow \mathrm{NSDTS} \Leftrightarrow \mathrm{RTS} \Rightarrow \mathrm{ERTS} \Leftrightarrow \mathrm{PTS}$$

Expansion Postponment is the conjecture that RTS $\Leftarrow$ ERTS. Now we ask whether $SDTS \Leftarrow NSDTS$, i.e. whether EP implies a system of syntax-directed derivation. For non-functional PTS this is false.

Notation: we write $\Gamma \vdash_{\rm nsd} M :\geq A$ for $\Gamma \vdash_{\rm nsd} M : A'$ and $A' \geq A$.

NSD-AX	$\bullet \vdash_{\rm nsd} c : s$	$\mathsf{Ax}(c{:}s)$
NSD-START	$\dfrac{\Gamma \vdash_{\rm nsd} A :\geq s}{\Gamma[x{:}A] \vdash_{\rm nsd} x : A}$	x *fresh*
NSD-WEAK	$\dfrac{\Gamma \vdash_{\rm nsd} \alpha : C \qquad \Gamma \vdash_{\rm nsd} A :\geq s}{\Gamma[x{:}A] \vdash_{\rm nsd} \alpha : C}$	x *fresh*
NSD-PI	$\dfrac{\Gamma \vdash_{\rm nsd} A :\geq s_1 \qquad \Gamma[x{:}A] \vdash_{\rm nsd} B :\geq s_2}{\Gamma \vdash_{\rm nsd} \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
NSD-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\rm nsd} M :\geq B \qquad \Gamma \vdash_{\rm nsd} \{x{:}A\}B : s}{\Gamma \vdash_{\rm nsd} [x{:}A]M : \{x{:}A\}B}$	
NSD-APP	$\dfrac{\Gamma \vdash_{\rm nsd} M :\geq \{x{:}A\}B \qquad \Gamma \vdash_{\rm nsd} N : A' \qquad A \simeq A'}{\Gamma \vdash_{\rm nsd} M\,N : [N/x]B}$	

Table 8: The nearly syntax-directed type system (NSDTS)

Notation: we write $\Gamma \vdash_{\rm nsd} M :\geq A$ for $\Gamma \vdash_{\rm sd} M : A'$ and $A' \geq A$.

SD-AX	$\bullet \vdash_{\rm sd} c : s$	$\mathsf{Ax}(c{:}s)$
SD-START	$\dfrac{\Gamma \vdash_{\rm nsd} A :\geq s}{\Gamma[x{:}A] \vdash_{\rm sd} x : A}$	x *fresh*
SD-WEAK	$\dfrac{\Gamma \vdash_{\rm sd} \alpha : C \qquad \Gamma \vdash_{\rm nsd} A :\geq s}{\Gamma[x{:}A] \vdash_{\rm sd} \alpha : C}$	x *fresh*
SD-PI	$\dfrac{\Gamma \vdash_{\rm nsd} A :\geq s_1 \qquad \Gamma[x{:}A] \vdash_{\rm nsd} B :\geq s_2}{\Gamma \vdash_{\rm sd} \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
SD-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\rm sd} M : B \qquad \Gamma \vdash_{\rm sd} \{x{:}A\}B : s}{\Gamma \vdash_{\rm sd} [x{:}A]M : \{x{:}A\}B}$	
SD-APP	$\dfrac{\Gamma \vdash_{\rm nsd} M :\geq \{x{:}A\}B \qquad \Gamma \vdash_{\rm sd} N : A' \qquad A \simeq A'}{\Gamma \vdash_{\rm sd} M\,N : [N/x]B}$	

Table 9: The syntax-directed type system (SDTS)

Example 3.2 *We show that NSDTS types terms that SDTS does not type, i.e.SDTS $\not\Leftarrow$ NSDTS, by modifying an example that Jutting [vBJ92] uses to show non-functional PTS may not have Subject Expansion.*

pathological	Cnst	$*, \triangle, \triangle', \Box$
	Sort	$*, \triangle, \triangle', \Box$
	Ax	$(*{:}\triangle), (*{:}\triangle'), (\triangle{:}\Box)$
	Rule	$(\Box, \Box, \Box,), (\triangle, \triangle', \Box)$

Consider $A = ([x{:}\triangle]*)\,*$. To compute all the types SDTS gives A, build the only possible derivation over A (since SDTS is syntax-directed!) and try to fill in any holes left un-determined due to the non-functionality of this pathological PTS. (We will use this technique more uniformly in Section 4.4.)

$$\dfrac{\dfrac{\dfrac{\vdots}{[x{:}\triangle] \vdash_{\mathrm{sd}} * : X} \qquad \dfrac{\bullet \vdash_{\mathrm{sd}} \triangle : \Box \qquad \dfrac{\vdots}{[x{:}\triangle] \vdash_{\mathrm{sd}} X : Z}}{\bullet \vdash_{\mathrm{sd}} \{x{:}\triangle\}X : \Box}}{\bullet \vdash_{\mathrm{sd}} [x{:}\triangle]* : \{x{:}\triangle\}X} \qquad \bullet \vdash_{\mathrm{sd}} * : \triangle \qquad \triangle \simeq \triangle}{\bullet \vdash_{\mathrm{sd}} A : [*/x]X}$$

First, $Z = \Box$ because only the first rule can be used to type $\{x{:}\triangle\}X$. Now observe that $X = \triangle$ or $X = \triangle'$ since these are the only types of $*$. But $(\triangle'{:}\Box) \notin$ Ax so $X = \triangle$ is the only solution, and we have only $\bullet \vdash_{\mathrm{sd}} A : \triangle$. (In fact, even in PTS this is the only solution. Since $\bullet \vdash A : \triangle$, $A \geq *$, $\bullet \vdash * : \triangle'$, but not $\bullet \vdash A : \triangle'$, this PTS lacks Subject Expansion.)

Now observe $[y{:}A] \vdash_{\mathrm{nsd}} [z{:}*]y : \{z{:}*\}*$

$$\dfrac{\dfrac{\dfrac{\vdots}{[y{:}A] \vdash_{\mathrm{nsd}} y : A} \qquad \dfrac{\vdots}{[y{:}A] \vdash_{\mathrm{nsd}} * : \triangle}}{[y{:}A][z{:}*] \vdash_{\mathrm{nsd}} y : A \geq *} \qquad \dfrac{\dfrac{\vdots}{[y{:}A] \vdash_{\mathrm{nsd}} * : \triangle} \qquad \dfrac{\vdots}{[y{:}A][z{:}*] \vdash_{\mathrm{nsd}} * : \triangle'}}{[y{:}A] \vdash_{\mathrm{nsd}} \{z{:}*\}* : \Box}}{[y{:}A] \vdash_{\mathrm{nsd}} [z{:}*]y : \{z{:}*\}*}$$

but SDTS does not assign any type to $[z{:}*]y$ in context $[y{:}A]$ as we see by trying to construct such a derivation

$$\dfrac{\dfrac{\vdots}{[y{:}A][z{:}*] \vdash_{\mathrm{sd}} y : A} \qquad \dfrac{\dfrac{\vdots}{[y{:}A] \vdash_{\mathrm{sd}} * : \triangle} \qquad \dfrac{\vdots}{[y{:}A][z{:}*] \vdash_{\mathrm{sd}} A : Y}}{[y{:}A] \vdash_{\mathrm{sd}} \{z{:}*\}A : X}}{[y{:}A] \vdash_{\mathrm{sd}} [z{:}*]y : \{z{:}*\}A}$$

By the analysis above, $Y = \triangle$, but there is no rule for $(\triangle, \triangle, \_)$, so there is no X making this a correct derivation.

This PTS is strongly normalizing, as can be seen from the PTS-morphism [Geu90]

$$* \mapsto \mathsf{Type}(0) \quad \triangle \mapsto \mathsf{Type}(1) \quad \triangle' \mapsto \mathsf{Type}(1) \quad \Box \mapsto \mathsf{Type}(2)$$

into a predicative part of ECC. Since this mapping preserves sorts, axioms, and rules, if a term is well-typed in pathological, its image is well-typed in ECC, hence strongly normalizing. But the map also preserves redices, so the original term is strongly normalizing. It is clearly the non-functionality of pathological that is the problem.

What is the Difficulty? The difficulty in proving EP and that in proving $SDTS \Leftarrow NSDTS$ are similar. In the LDA rule, the predicate of the left premiss, B, occurs in the subject of the right premiss, $\{x{:}A\}B$. This is the only rule in which this happens. For any of the systems above without full conversion we cannot derive all types of all terms; the best correctness property we can have allows some extra conversion or reduction after a derivation. In LDA, then, the B occurring in the left premiss will be different than the B occurring in the right premiss. If we could prove subject reduction of RTS the proof would go through, but the proof of subject reduction for RTS founders on the same difficulty. We have seen that $SDTS \Leftarrow NSDTS$ must fail in general. (I don't know whether it is true for functional PTS.) Is this a hint that EP fails in general? If so, is EP true for functional PTS?

Recently Jutting has been looking at alternative "well-formedness" conditions to replace the right premiss of LDA. He has had great sucess in finding syntax-directed systems, which can be made into typechecking algorithms. I do not know whether this sheds any light on EP.

4 Semi-Full PTS and Typechecking

A PTS is *full* iff for all $s_1, s_2 \in \mathsf{Sort}$ there exists $s_3 \in \mathsf{Sort}$ such that $\mathsf{Rule}(s_1, s_2, s_3)$. In full PTS the troublesome right premiss of the LDA rule can be simplified. Consider the rule

$$\text{LDA} \qquad \frac{\Gamma[x{:}A] \vdash M : B \qquad \Gamma \vdash \{x{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$$

The right premiss guarantees the type of the lambda being constructed is itself well formed. We know from the left premiss of LDA that $\Gamma \vdash A : s_A$ for some s_A. Also, by Correctness of Types applied to the left premiss there is some s_B such that either $\Gamma[x{:}A] \vdash B : s_B$ or $B = s_B$. In the first case fullness of a PTS guarantees $\{x{:}A\}B$ has some type. Thus, for full PTS, we can soundly replace the right premiss of LDA with a side condition that B is not identically a sort that itself has no sort. We can generalize this idea somewhat beyond full PTS:

Definition 4.1 A PTS is *semi-full* iff for all $s_1 \in \mathsf{Sort}$, if there exist $s_2, s_3 \in \mathsf{Sort}$ such that $\mathsf{Rule}(s_1, s_2, s_3)$, then for all $s_2 \in \mathsf{Sort}$ there exist $s_3 \in \mathsf{Sort}$ such that $\mathsf{Rule}(s_1, s_2, s_3)$.

While the Pure Calculus of Constructions, $\lambda P\omega$, and various extensions with Type universes are full, the Edinburgh Logical Framework, λP, is semi-full. Although [HHP92] proves LF is decidable, it gives an algorithm to compute the normal forms of types. The argument below shows why the algorithm actually used in LEGO, and probably in ELF, is both sound and complete.

We will give a syntax-directed derivation system for semi-full PTS, and from it derive an algorithm for typechecking which is a natural generalization of Huet's Constructive Engine.

4.1 Aside on Topsorts

We begin with a definition and a lemma of Berardi [Ber90].

Definition 4.2 A sort s is a *topsort* (written $s \in \mathsf{Sort}^T$) iff for all $t \in \mathsf{Sort}$, not $\mathsf{Ax}(s{:}t)$.

Lemma 4.1 *If $s \in \mathsf{Sort}^T$ and $\Gamma \vdash M : A$ then*
(1) s does not occur in M, and
(2) if s occurs in A then $s = A$.

$$\text{SF-AX} \qquad \bullet \vdash_{\mathsf{sf}} c : s \qquad \mathsf{Ax}(c{:}s)$$

$$\text{SF-START} \qquad \frac{\Gamma \vdash_{\mathsf{sf}} A : s}{\Gamma[x{:}A] \vdash_{\mathsf{sf}} x : A} \qquad x\ \textit{fresh}$$

$$\text{SF-WEAK} \qquad \frac{\Gamma \vdash_{\mathsf{sf}} \alpha : C \qquad \Gamma \vdash_{\mathsf{sf}} A : s}{\Gamma[x{:}A] \vdash_{\mathsf{sf}} \alpha : C} \qquad x\ \textit{fresh}$$

$$\text{SF-PI} \qquad \frac{\Gamma \vdash_{\mathsf{sf}} A : s_1 \qquad \Gamma[x{:}A] \vdash_{\mathsf{sf}} B : s_2}{\Gamma \vdash_{\mathsf{sf}} \{x{:}A\}B : s_3} \qquad \mathsf{Rule}(s_1, s_2, s_3)$$

$$\text{SF-LDA} \qquad \frac{\Gamma[x{:}A] \vdash_{\mathsf{sf}} M : B \qquad \Gamma \vdash_{\mathsf{sf}} A : s}{\Gamma \vdash_{\mathsf{sf}} [x{:}A]M : \{x{:}A\}B} \qquad \begin{array}{c} B \notin \mathsf{Sort}^T \\ \mathsf{Rule}(s, s_2, s_3) \end{array}$$

$$\text{SF-APP} \qquad \frac{\Gamma \vdash_{\mathsf{sf}} M : \{x{:}A\}B \qquad \Gamma \vdash_{\mathsf{sf}} N : A}{\Gamma \vdash_{\mathsf{sf}} M\,N : [N/x]B}$$

$$\text{SF-CONV} \qquad \frac{\Gamma \vdash_{\mathsf{sf}} M : A \qquad \Gamma \vdash_{\mathsf{sf}} B : s \qquad A \simeq B}{\Gamma \vdash_{\mathsf{sf}} M : B}$$

Table 10: The typing judgement of a semi-full PTS (SFTS)

Prove (1) by straightfoward induction over the derivation of $\Gamma \vdash M : A$. Prove (2) similarly, using (1).

The Correctness of Types lemma for PTS is somewhat unsatisfying: why doesn't it read "If $\Gamma \vdash b : B$ then for some $s \in \mathsf{Sort}$ $\Gamma \vdash B : s$"? The reason is the existence of topsorts, and we would like to improve Correctness of Types to read "If $\Gamma \vdash b : B$ then for some $s \in \mathsf{Sort}$ $\Gamma \vdash B : s$ or ($B = s$ and $s \in \mathsf{Sort}^T$)". This is not quite right, because even if Ax is decideable, Sort^T may not be. Furthermore, topsort is a negative notion; we need the more informative concept

Definition 4.3 (1) A sort, s, is a *typedsort* (written $s \in \mathsf{Sort}_T$) iff there exists $t \in \mathsf{Sort}$ such that $\mathsf{Ax}(s{:}t)$.
(2) A PTS has the *decidable typedsort property* (DTP) iff for all $s \in \mathsf{Sort}$, $s \in \mathsf{Sort}_T$ or $s \notin \mathsf{Sort}_T$.

Lemma 4.2 *If a PTS has DTP and $\Gamma \vdash b : B$ then $\exists s \in \mathsf{Sort}$ such that $\Gamma \vdash B : s$ or ($s \in \mathsf{Sort}^T$ and $B = s$).*

Proof By the Correctness of Types lemma, $\Gamma \vdash B : s$ or $B = s$. In the first case we are done. In the second case, by DTP, $s \in \mathsf{Sort}_T$ or $s \notin \mathsf{Sort}_T$. In the first case, for some t, $\mathsf{Ax}(s{:}t)$; hence $\Gamma \vdash B : t$. In the second case $s \in \mathsf{Sort}^T$ as required. ∎

4.2 Deriving the Judgement of Semi-Full PTS

For semi-full PTS with DTP, the derivation system SFTS of Table 10 is sound and complete with respect to the system PTS of Table 5.

Lemma 4.3 *For a semi-full PTS with DTP*
(1) if $\Gamma \vdash M : A$ *then* $\Gamma \vdash_{\mathrm{sf}} M : A$, *and*
(2) if $\Gamma \vdash_{\mathrm{sf}} M : A$ *then* $\Gamma \vdash M : A$.

Proof (1) By induction over the derivation of $\Gamma \vdash M : A$. The only interesting case is LDA, so assume the derivation ends with

$$\text{LDA} \qquad \frac{\begin{array}{c}\vdots\\ \Gamma[x{:}A] \vdash M : B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash \{x{:}A\}B : s\end{array}}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$$

By IH $\Gamma[x{:}A] \vdash_{\mathrm{sf}} M : B$ and $\Gamma \vdash_{\mathrm{sf}} \{x{:}A\}B : s$. By the Generation Lemma for SFTS, there exists $s_A, s_B, s_3 \in \mathsf{Sort}$ such that $\Gamma \vdash_{\mathrm{sf}} A : s_A$, $\Gamma[x{:}A] \vdash_{\mathrm{sf}} B : s_B$, and $\mathsf{Rule}(s_A, s_B, s_3)$. Lemma 4.1 on the right IH shows $B \notin \mathsf{Sort}^T$. Thus, by SF-LDA conclude $\Gamma \vdash_{\mathrm{sf}} [x{:}A]M : \{x{:}A\}B$.

(2) By induction over the derivation of $\Gamma \vdash_{\mathrm{sf}} M : A$. The only interesting case is SF-LDA, so assume the derivation ends with

$$\text{SF-LDA} \qquad \frac{\begin{array}{c}\vdots\\ \Gamma[x{:}A] \vdash_{\mathrm{sf}} M : B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash_{\mathrm{sf}} A : s\end{array}}{\Gamma \vdash_{\mathrm{sf}} [x{:}A]M : \{x{:}A\}B} \qquad \begin{array}{r} B \notin \mathsf{Sort}^T \\ \mathsf{Rule}(s, s_2, s_3)\end{array}$$

By IH $\Gamma[x{:}A] \vdash M : B$ and $\Gamma \vdash A : s$. By Lemma 4.2 on the left IH and the fact that $B \notin \mathsf{Sort}^T$, there is a sort s_B such that $\Gamma[x{:}A] \vdash B : s_B$. Since we are in a semi-full PTS, there is a sort t with $\mathsf{Rule}(s, s_B, t)$. By PI and LDA conclude $\Gamma \vdash [x{:}A]M : \{x{:}A\}B$. ∎

4.3 A Syntax-Directed System for Semi-Full PTS

System SFTS is not syntax-directed, but SF-LDA does not have the troublesome right premiss. It is now straightfoward to give a syntax-directed system, SDSFTS of Table 11, which is sound and complete for SFTS.

Lemma 4.4 *For a semi-full PTS with DTP:*
(1) if $\Gamma \vdash_{\mathrm{sdsf}} M : A$ *then* $\Gamma \vdash_{\mathrm{sf}} M : A$,
(2) if $\Gamma \vdash_{\mathrm{sf}} M : A$ *then there exists* A' *s.t.* $\Gamma \vdash_{\mathrm{sdsf}} M : A'$ *and* $A \simeq A'$.

Proof (1) By induction over the derivation of $\Gamma \vdash_{\mathrm{sdsf}} M : A$, noticing that by Lemma 4.3 SFTS has Predicate Reduction for semi-full PTS with DTP. We do the case where the derivation ends with SDSF-APP

$$\text{SDSF-APP} \qquad \frac{\begin{array}{c}\vdots\\ \Gamma \vdash_{\mathrm{sdsf}} M :\geq \{x{:}A\}B\end{array} \qquad \begin{array}{c}\vdots\\ \Gamma \vdash_{\mathrm{sdsf}} N : A'\end{array} \qquad A \simeq A'}{\Gamma \vdash_{\mathrm{sdsf}} M\, N : [N/x]B}$$

By IH $\Gamma \vdash_{\mathrm{sf}} M : K$ where $K \geq \{x{:}A\}B$, and $\Gamma \vdash_{\mathrm{sf}} N : A'$ where $A \simeq A'$. By Church-Rosser A and A' have a common reduct, A''. Thus $K \geq \{x{:}A''\}B$, and by Predicate Reduction $\Gamma \vdash_{\mathrm{sf}} M : \{x{:}A''\}B$. By SF-APP conclude $\Gamma \vdash_{\mathrm{sf}} M\, N : [N/x]B$ as required.

SDSF-AX	$\bullet \vdash_{\text{sdsf}} c : s$	$\mathsf{Ax}(c{:}s)$
SDSF-START	$\dfrac{\Gamma \vdash_{\text{sdsf}} A :\geq s}{\Gamma[x{:}A] \vdash_{\text{sdsf}} x : A}$	x *fresh*
SDSF-WEAK	$\dfrac{\Gamma \vdash_{\text{sdsf}} \alpha : C \qquad \Gamma \vdash_{\text{sdsf}} A :\geq s}{\Gamma[x{:}A] \vdash_{\text{sdsf}} \alpha : C}$	x *fresh*
SDSF-PI	$\dfrac{\Gamma \vdash_{\text{sdsf}} A :\geq s_1 \qquad \Gamma[x{:}A] \vdash_{\text{sdsf}} B :\geq s_2}{\Gamma \vdash_{\text{sdsf}} \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
SDSF-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\text{sdsf}} M : B \qquad \Gamma \vdash_{\text{sdsf}} A :\geq s}{\Gamma \vdash_{\text{sdsf}} [x{:}A]M : \{x{:}A\}B}$	$B \notin \mathsf{Sort}^T$ $\mathsf{Rule}(s, s_2, s_3)$
SDSF-APP	$\dfrac{\Gamma \vdash_{\text{sdsf}} M :\overset{wh}{\geq} \{x{:}A\}B \qquad \Gamma \vdash_{\text{sdsf}} N : A' \qquad A \simeq A'}{\Gamma \vdash_{\text{sdsf}} M\,N : [N/x]B}$	

Table 11: The syntax-directed semi-full type system (SDSFTS)

(2) By induction over the derivation of $\Gamma \vdash_{\text{sf}} M : A$. We do one case. Assume the derivation ends with SF-LDA

$$\text{SF-LDA} \qquad \dfrac{\begin{matrix}\vdots & & \vdots\\ \Gamma[x{:}A] \vdash_{\text{sf}} M : B & \qquad & \Gamma \vdash_{\text{sf}} A : s\end{matrix}}{\Gamma \vdash_{\text{sf}} [x{:}A]M : \{x{:}A\}B} \qquad \begin{matrix} B \notin \mathsf{Sort}^T \\ \mathsf{Rule}(s, s_2, s_3)\end{matrix}$$

By IH $\Gamma[x{:}A] \vdash_{\text{sdsf}} M : B'$ where $B' \simeq B$, and $\Gamma \vdash_{\text{sdsf}} A :\geq s$. In order to conclude $\Gamma \vdash_{\text{sdsf}} [x{:}A]M : \{x{:}A\}B'$ by SDSF-LDA it remains to show $B' \notin \mathsf{Sort}^T$. If $B' = t \in \mathsf{Sort}^T$ then $B \geq t$ so t occurs in B. Notice that $\Gamma[x{:}A] \vdash M : B$ by Lemma 4.3 (2) applied to the left premiss, hence $B = t$ by Lemma 4.1; but this contradicts the given side condition. ∎

Finally, we could apply the optimization, discussed in Section 1.2, which avoids duplicating the context validity check. Since this is straightfoward I forego this extra step.

4.4 A Typechecking Algorithm for Semi-Full PTS

SDSFTS of Table 11 is syntax directed. In the case of a normalizing, functional PTS with finite Sort we can already view SDSFTS as a type synthesis algorithm in straightfoward generalization of the PCC example of Section 1.2. (Thus, for LF, we are done.) In general two more problems arise: in systems with infinite Sort it may be undecidable if there is an axiom for a given constant c, or a rule for given s_1, s_2; in non-functional systems we won't know which of several axioms for c to choose, or which of several rules for s_1, s_2. These problems are reduced to a decision problem about the sorts, axioms and rules of the PTS by making these decisions schematically and collecting the conditions constraining the schematic choices. This subsection follows [HP91], where more details of a related application of the same ideas can be found. The same technique can be used to derive algorithms from syntax-directed derivation systems for other classes of type theory, such as the systems of Jutting.

Notation: we write $\Gamma \vdash_{\mathrm{ds}} M \Rightarrow\geq X, \mathcal{C}$ for $\Gamma \vdash_{\mathrm{ds}} M \Rightarrow X', \mathcal{C}$, $\mathcal{C}$ consistent, and $X' \geq X$.

DS-AX	$\bullet \vdash_{\mathrm{ds}} c \Rightarrow \sigma, \{\mathsf{Ax}(c{:}\sigma)\}$	σ *fresh*
DS-START	$\dfrac{\Gamma \vdash_{\mathrm{ds}} A \Rightarrow\geq \sigma, \mathcal{C}}{\Gamma[x{:}A] \vdash_{\mathrm{ds}} x \Rightarrow A, \mathcal{C}}$	x *fresh*
DS-WEAK	$\dfrac{\Gamma \vdash_{\mathrm{ds}} \alpha \Rightarrow X, \mathcal{C} \qquad \Gamma \vdash_{\mathrm{ds}} A \Rightarrow\geq \sigma, \mathcal{D}}{\Gamma[x{:}A] \vdash_{\mathrm{ds}} \alpha \Rightarrow X, \mathcal{C} \cup \mathcal{D}}$	x *fresh*
DS-PI	$\dfrac{\Gamma \vdash_{\mathrm{ds}} A \Rightarrow\geq \sigma_1, \mathcal{C} \qquad \Gamma[x{:}A] \vdash_{\mathrm{ds}} B \Rightarrow\geq \sigma_2, \mathcal{D}}{\Gamma \vdash_{\mathrm{ds}} \{x{:}A\}B \Rightarrow \sigma_3, \mathcal{C} \cup \mathcal{D} \cup \{\mathsf{Rule}(\sigma_1, \sigma_2, \sigma_3)\}}$	σ_3 *fresh*
DS-LDA	$\dfrac{\Gamma[x{:}A] \vdash_{\mathrm{ds}} M \Rightarrow X, \mathcal{C} \qquad \Gamma \vdash_{\mathrm{ds}} A \Rightarrow\geq \sigma, \mathcal{D}}{\Gamma \vdash_{\mathrm{ds}} [x{:}A]M \Rightarrow \{x{:}A\}B, \mathcal{C} \cup \mathcal{D} \cup \{B \notin \mathsf{Sort}^T\} \cup \{\mathsf{Rule}(\sigma, \sigma_2, \sigma_3)\}}$	σ_2, σ_3 *fresh*
DS-APP	$\dfrac{\Gamma \vdash_{\mathrm{ds}} M \Rightarrow\overset{wh}{\geq} \{x{:}X\}Y, \mathcal{C} \quad \Gamma \vdash_{\mathrm{ds}} N \Rightarrow X', \mathcal{D} \quad \mathcal{C} \cup \mathcal{D} \text{ consistent} \quad X \overset{\mathcal{E}}{\simeq} X'}{\Gamma \vdash_{\mathrm{ds}} M\,N \Rightarrow [N/x]Y, \mathcal{C} \cup \mathcal{D} \cup \mathcal{E}}$	

Table 12: Derivation schemes for the syntax-directed semi-full type system (SDSFDS)

Recall the discussion of schematic terms in Section 2.3. For our current purposes a *constraint* is one of the expressions $\mathsf{Ax}(c{:}\sigma)$, $\mathsf{Rule}(\sigma_1, \sigma_2, \sigma_3)$, $\sigma_1 = \sigma_2$, or $B \notin \mathsf{Sort}^T$. We henceforth assume that the consistency of any set of constraints is decidable (notice that this implies DTP). For example, if Sort is finite (as in LF) this is clearly the case; a system such as LF extended with infinitely many stratified universes also has this property.

4.4.1 Schematic Typing

Table 12 is a system for *derivation schemes* of the system SDSFTS. The judgements of SDSFDS have the shape $\Gamma \vdash_{\mathrm{ds}} M \Rightarrow X, \mathcal{C}$. All the constraints that must be satisfied for an instance of X to be a correct type of M in Γ are collected in $\mathcal{C}$.

Lemma 4.5 *For a semi-full PTS with consistency of constraint sets decidable, if* $\Gamma \vdash_{\mathrm{ds}} M \Rightarrow X, \mathcal{C}$ *then* $X, \mathcal{C}$ *is a mgts for* Γ, M.

SDSFDS has an algorithmic interpretation returning failure if Γ, M has no type, or an mgts for Γ, M

References

[Bar91] Henk Barendregt. Introduction to generalised type systems. *J. Functional Programming*, 1(2):124–154, April 1991.

[Bar92] Henk Barendregt. Lambda calculii with types. In Gabbai Abramsky and Maibaum, editors, *Handbook of Logic in Computer Science*, volume II. Oxford University Press, 1992.

[Ber90] Stefano Berardi. *Type Dependence and Constructive Mathematics.* PhD thesis, Dipartimento di Informatica, Torino, Italy, 1990.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.

[Geu90] Herman Geuvers. Type systems for higher order logic. 1990.

[GN91] Herman Geuvers and Mark-Jan Nederhof. A modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, April 1991.

[Hel91] Leen Helmink. Goal directed proof construction in type theory. In *Logical Frameworks*. Cambridge University Press, 1991.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science*, pages 194–204, Ithaca, New York, June 1987.

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 1992. to appear.

[HP91] Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

[Hue87] Gérard Huet. The constructive engine, March 1987. Invited talk at ESOP'88. Not in proceedings.

[LP92] Zhaohui Luo and Robert Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, Computer Science Dept., University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, May 1992. Updated version.

[Luo90] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified mataprogramming. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science, Asilomar, California*, June 1989.

[vBJ92] L.S. van Bentham Jutting. Typing in pure type systems. *Information and Computation*, 1992. To appear.

A Relevant Analysis of Natural Deduction

David Pym and Gordon Plotkin, Edinburgh

Abstract

We discuss a first-order dependent type theory based on a fragment of intuitionistic linear logic. We discuss the use of this type theory as a framework for the analysis of natural deduction presentations of weak logics.

Fixed point and type systems

Christophe Raffalli, Paris 7

Abstract

We study an extension of AF2 type system (system F + first order + equations). This extension adds two new quantifiers on formulas: least fixpoint (inductive types) and greatest fixpoint (co-inductive types). We denote them respectively by mu and nu.

In this system we prove the following results :

1. We have found conditions on types and proofs which ensure that all typed terms are hereditarily solvable (their Bohm-tree don't contain any bottom). The surprising fact is that the condition is stronger with least fixpoint (inductive types) than for the system with only greatest fixpoint.

   The condition for the system with only greatest fixpoint is that there is nowhere in the proof a subformula "nu X.F" where X is strictly positive in F (in particular no "nu X.X" in the proof). Concerning the least fixpoint, the condition concern all subformulas in the proof and have the same complexity than decision of propositional classical logic.

2. We have also found in this system, a representation for inductive and co-inductive data. This types are such that each data has a unique normal representation (in fact a unique Bohm-tree). For instance, this allows a construction of the data-type of streams, ensuring unicity of the representation of each stream.

3. We can build a non-standard representation for natural numbers using the greatest fixpoint, with which one can encode all partial recursive functions.

An Overview of the MIZAR Project

Piotr Rudnicki*
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2H1

e-mail: piotr@cs.ualberta.ca

June 30, 1992

Abstract

The MIZAR project is a long-term effort aimed at developing software to support a working mathematician in preparing papers. A. Trybulec, the leader of the project, has designed a language for writing formal mathematics. The logical structure of the language is based on a natural deduction system developed by Jaśkowski. The texts written in the language are called MIZAR articles and are organized into a data base. The Tarski-Grothendieck set theory forms the basis of doing mathematics in MIZAR. The implemented processor of the language checks the articles for logical consistency and correctness of references to other articles.

1 Introduction

The idea that an automatic device should check our logical derivations is by no means new. It can be traced back not only to Pascal and Leibniz, but to Ramon Llull. In recent years, several projects have aimed at providing computer assistance for doing mathematics. Among the better known there are: AUTOMATH [4], EKL [10], QUIP [22], Nuprl [3], THEAX [14], Computational Logic [2], Ontic [11], and the more recent ones such as ALF, ELF, HOL, LEGO and many others (see [6, 7]). The specific goals of these projects vary. However, they have one common feature: the human writes mathematical texts and the machine verifies their correctness.

The project MIZAR started almost 20 years ago under the leadership of Andrzej Trybulec at the Płock Scientific Society, Poland. Its original goal was to design and implement a software environment to assist the process of preparing mathematical papers. For lack of a better alternative, the project was based upon the style of doing mathematics used by the mathematicians of the so-called Polish mathematical school. Therefore, the project can be seen as an attempt to develop software environment for writing traditional mathematical papers, where classical logic and set theory form the basis of all future developments.

The logical basis of the system is a "Polish" style of natural deduction. Only after years of using the logic has it been learned that it was the "composite system of logic" developed by Stanisław Jaśkowski, see [8] and [12] for the English translation. Katuzi Ono, [16] described a

*This work was supported in part by NSERC Grant OGP9207.

similar system. Various formalizations of set theory have been tried (Zermelo-Fraenkel, Morse-Kelley) but finally the Tarski-Grothendieck axiomatization has been adopted.

2 A bit of history

The name Mizar[1] was picked up in 1973 for a different project (a programming environment) that was discontinued, but its name has been recycled.

The first experiments in 1974-75 developed a modest proof-checker for propositional logic. The main concern of A. Trybulec was the input language to the checker, and it turned out to be the main concern for all future years. The proof checker was based on a fixed set of inference rules. The justification of an inference in the input text required the user to state only the premises and the conclusion; the checker searched for a rule or a sequence of rules to validate an inference step. This approach was abandoned, and all future MIZAR processors have used model checking.

In 1977 the language and the checker were extended with quantifiers to form MIZAR QC which had neither functional notation nor definitional facilities. These were added in subsequent years to form the MIZAR FC system, which was used to record a number of larger texts. Among these texts was the initial segment of the book on arithmetics by Grzegorczyk [5]. The book is so rigorous and detailed (the Chinese remainder theorem comes as late as page 67 and its proof takes 6 pages) that the blow-up factor in the translation to MIZAR FC was negligible.

At around 1978-1979 the MIZAR group started to grow substantially, being anchored at Warsaw University, Białystok[2] Branch[3].

In 1981, a language called MIZAR 2 had been designed by A. Trybulec and implemented on an ICL 1900 by Cz. Byliński, H. Oryszczyszyn, P. Rudnicki, and A. Trybulec. The system was written in Pascal and later ported to other computers (mainframe IBM and also to UNIX). Again, the stress was on the input language. The proof checker was rather weak which forced one to write very detailed proofs. The language included the following features: structured types, type hierarchy, comprehensive definitional facilities, built-in fragments of arithmetics, and a built-in variant of set theory. The translations into MIZAR 2 included: a number of recent papers in topology, projective geometry, and some text-book algebra. In one of the experiments, the pigeon-hole principle was proven from scratch and compared with the similar development by van Benthem Jutting [28] in AUT-QE (the AUTOMATH project). The MIZAR 2 text was about half as long as the AUT-QE text; however, MIZAR 2 and AUT-QE were substantially different environments for conducting proofs (logic based vs type based). Among other works with MIZAR 2, there were attempts to prove properties of programs [21] and software specifications [20].

In the following years, other MIZAR languages and their implementations have been developed but their character was experimental (MIZAR 3, MIZAR HPF); the systems were not distributed outside the MIZAR group in Białystok, with one exception.

[1]Mizar, ζ Ursae Majoris, is the second magnitude star set in the middle of the Big Dipper's handle; Mizar (Arabic: veil, cloak, burial grounds) makes a visual binary with the fainter Alcor (Arabic: faint one); each of the visual components is a spectroscopic binary; Mizar is a quadruple star.

Incidentally, Algol (Arabic: daemon, ghoul), β Persei, is an eclipsing visual binary star, spectroscopically a triple, probably a quadruple star as well.

[2]E. Post was born in Augustów, near Białystok. A. Lindenbaum was last seen in Białystok in 1941.

[3]Address: Mizar Group, Institute of Mathematics, Warsaw University—Białystok Branch, 15-276 Białystok, Poland.

A subset of MIZAR, named MIZAR MSE (short for Multi-Sorted with Equality) was implemented in 1982 by R. Matuszewski, P. Rudnicki, and A. Trybulec and has been widely used since then. The system is meant for teaching elementary logic with emphasis on the practical aspects of constructing proofs. The MIZAR MSE language encompasses 'raw' predicate calculus, multi-sorted with equality, but the language does not provide functional notation or special notation for definitions. There are numerous implementations of MIZAR MSE, see [26, 25, 13, 19, 18, 15].

In 1986, MIZAR 4 was implemented as a redesign of MIZAR 2 and distributed to several dozen users. Each MIZAR 4 article included a preliminaries part where the author could state some axioms that were not checked for validity.

In 1988 the design of the language was completed by A. Trybulec and the final language is named simply MIZAR. While articles in previous versions of the language must be self-contained, the final MIZAR allows for cross-references among articles. Moreover, an author of a MIZAR text is not allowed to introduce new axioms. Only the predefined axioms can be used, everything else must be proved. The two articles that are not checked for validity contain an axiomatics of the Tarski-Grothendieck set theory (see Appendix A) and definitional axioms of the built-in concepts and axioms of strong arithmetics of real numbers (see Appendix B).

Recently, the main effort in the MIZAR project has been in building the library of MIZAR articles which now numbers almost 300.

The development of MIZAR has been driven by "experience, not only doctrine (ENOD for short)"[4]. In this case it meant that any idea that made sense was first of all implemented and tried by its proponent. However, only after other users approved the idea by actually using its implementation, was the idea included into the state of the project. The ENOD approach has been applied in the development of the MIZAR processor, the input language (although it is almost exclusively of A. Trybulec's creation), and the MIZAR library.

Before giving more information about MIZAR it may be worthwhile to recall here the *Formulaire de mathématiques* project of Giuseppe Peano (quoted from Kennedy [9] p. 8):

> The end result of this project would be, he hoped, the publication of a collection of all known theorems in the various branches of mathematics. The notions of his mathematical logic were to be used and proofs of the theorems were to be given. There were five editions of the *Formulario*: the first appeared in 1895, and the last, completed in 1908, contained some 4200 theorems.

The project was based on a formal language. In the introduction to the second volume of *Formulaire de mathématiques*, Peano writes (quoted from [17], p. 197, vol. II):

> Dans le petit livre ≪Arithmetices principia, nova methodo exposita, a. 1889≫, nous avons pour la première fois exposé toute une théorie, théorèmes, définitions et démonstrations, en symboles qui remplacent tout-à-fait le langage ordinaire.
>
> Nous avons donc la solution du problème proposé par Leibniz.

3 Anatomy of a Mizar article

Each MIZAR article is written as a text file. The general structure of such an article is as follows:

[4]The expression has been coined by G. Kreisel, see his contribution in *Logic and Computer Science*, P. Odifreddi (ed.), Academic Press, pp. 205–278.

```
environ
                Environment directives
begin
                Text-Proper Section
                . . .
                . . .
begin
                Text-Proper Section
```

The *Text-Proper* contains statements of facts with their proofs and definitions of new concepts with justification of their correctness. The *Environment directives* declare which items of the MIZAR library can be referenced from the *Text-Proper*. The directive

`vocabulary` *Vocabulary-File-Name*;

adds the symbols introduced in the *Vocabulary-File-Name* to the article's lexicon. Vocabulary files introduce new symbols of MIZAR expression constructors. The vocabularies also indicate the binding strength of the introduced symbols for parsing purposes. The authors can use existing vocabularies (there are hundreds of symbols there) and also are free to create new ones. The lexical make-up of new symbols is governed by a small set of quite liberal rules. The authors can freely enhance the zoo of mathematical symbols with new symbols of their own design.

One vocabulary is automatically attached to every MIZAR article. It introduces the following symbols: `Element`, `Subset`, `DOMAIN`, `Real`, `Nat`, `+`, `<>`, $\leq$, $\in$, and the like. The information pertaining to the usage of these symbols is built into the MIZAR processor, e.g. $\in$ can be used as a binary, infix predicate symbol with both arguments expandable to type `set`. Similarly, $\leq$ requires two arguments of type expandable to `Element of Real`.

Besides vocabularies, there are four kinds of data base directives:

`signature` *Signature-File-Name*;
`definitions` *Definitions-File-Name*;
`theorems` *Theorems-File-Name*;
`schemes` *Schemes-File-Name*;

The directive `signature` informs the MIZAR processor that the article is permitted to use the notation (definienda) introduced in article *Signature-File-Name*.

Any article can define ways in which the symbols contained in vocabularies can be used to form MIZAR expressions. Each of the MIZAR expression constructors (functor, predicate, mode) can be syntactically defined in a number of formats. These constructors may take various numbers of arguments of various types, and in the case of functors they may return results of various types. This creates a complicated system of overloaded constructors. The information needed for parsing is kept in auxiliary files associated with every article and commonly referred to as the signature of the article.

The remaining three directives allow us to use definitions, theorems, and schemes that are defined or proved in another article.

The *Text-Proper* is a sequence of sections, each being a sequence of *Text-Item*. There are the the following kinds of items:

- *Reservation* is used to reserve identifiers for a type. If a variable has an identifier reserved for a type, and no explicit type is stated for the variable, then the variable type defaults to the type for which its identifier was reserved.

- *Definition-Block* is used to define (or redefine) constructors of MIZAR phrases: term constructors (functions), formula constructors (predicates), and type constructors (modes).

- *Structure-Definition* introduces new structures. A structure is an entity that consists of a number of fields that are accessed by selectors.

- *Theorem* announces a proposition that can be referenced from other articles.

- *Scheme* also announces a proposition, visible from outside. Second order terms can occur in *scheme*.

- *Auxiliary-Item* introduces objects that are local to the article in which they occur and are not exported to the library files (e.g. lemmas, definitions of local predicates).

The goal of writing an article is to prove some theorems and schemes or to define some new concepts such that they can be referenced by other authors. Before the theorems and definitions are included into the library they must be proved valid and correct.

4 The PC Mizar

PC MIZAR is a MIZAR processor implemented on IBM PCs under DOS by Cz. Byliński, A. Trybulec, and S. Żukowski, and now further developed by the first two authors.

The central concept of MIZAR is a MIZAR *article*. Such an article can be viewed as an extremely detailed mathematical text written in a fixed formal notation, originally as a text file. There are rather few interesting things that one can prove in a short MIZAR article without making references to other articles. Usually, we base our work on the achievements of others.

The power of the MIZAR system is in its automatic processing of cross-references among articles contained in the MIZAR library. In order to speed up the process of cross reference checking, some internal files, derived from the submitted articles, are maintained. These files (they are not meant to be read by humans) are created in the process of including an article into the MIZAR library.

- *signature files* that for each newly defined constructor of MIZAR phrases in the article give information necessary for parsing the constructor occurrences.

- *definitions file* stores the definiens of every definition in the article, the definiendum is stored in the signature file.

- *theorems file* stores the theorems proved in the article (without proofs).

- *schemes file* stores the schemes proved in the article (without proofs).

The MIZAR software is a collection of about 50 programs that process MIZAR articles.

- The verifier must run in the appropriate environment with access to all the vocabulary and library files referenced in the given article. For efficiency reasons, each checked article obtains a dedicated environment (by a program called *accommodator*) in order to avoid too many references to different library files.

- Parsing of MIZAR texts is relatively complicated mainly because of the rich MIZAR syntax, multi-way overloading of names, and new definitions or redefinitions of MIZAR phrase constructors and their priorities.
- Checking whether proofs are correctly structured requires some processing as MIZAR permits a multitude of proof structures in the spirit of natural deduction proposed by Jaśkowski.
- An inference of the form

  $$\textit{premise}_0,\ \textit{premise}_1,\ \ldots,\ \textit{premise}_k\ \vdash\ \textit{conclusion}$$

  is transformed into the conjunction

  $$\textit{premise}_0\ \&\ \textit{premise}_1\ \&\ \ldots\ \&\ \textit{premise}_k\ \&\ \texttt{not}\ \textit{conclusion}.$$

  If the checker finds the conjunction contradictory then the original inference is accepted. Unfortunately but inevitably, the checker sometimes does not accept an inference that is logically correct; to get the inference accepted one has to split it into a sequence of 'smaller' ones, or possibly use a proof structure. The stress in the inference checker is on the processing speed, not power.

5 The Input Language

Experience has shown that people with even minimal mathematical training develop a good idea about the nature of the MIZAR language just by browsing through a sample article. This is not a big surprise as one of the original goals of the project was to build an environment that mimics the traditional ways that mathematicians work. A sample MIZAR article is presented in Appendix C.

Because of the richness of the MIZAR grammar, even a sketchy presentation of it is far beyond the scope of this text.

6 Mizar abstracts

The source texts of MIZAR articles tend to be lengthy as they contain complete proofs in a rather demanding formalism. New articles strongly depend on already existing ones. Therefore, there was a need to provide authors with a quick reference to the already collected articles. The solution was to automatically create an *abstract* for each MIZAR article. Such an abstract includes a presentation of all the items that can be referenced from other articles. The abstract of the article presented in Appendix C is contained in Appendix D. Therefore, there is no need to examine the entire article to make a reference to a single theorem.

To make the abstracts resemble a mathematical paper at least at the lexical level, they are automatically typeset using TEX. The TEXed MIZAR abstract from Appendix D is in Appendix E.

The typeset MIZAR abstracts are periodically published[5] by Université Catholique de Louvain as *Formalized Mathematics (a computer assisted approach)* with R. Matuszewski as editor.

[5] For more information write to: Fondation Philippe le Hodey, MIZAR Users Group, Av. F. Roosevelt 134 (Bte7), 1050 Brussels, Belgium, fax +32 (2) 640 89 68

7 Main Mizar Library

At the beginning of 1989, the MIZAR group in Białystok started collecting MIZAR articles and organizing them into a library that is distributed to other MIZAR users.

The person responsible for the library (E. Woronowicz) requires that authors of contributed articles supply an additional file that describes the bibliographic data such as title, authors' names and affiliations, and a summary (in English). The bibliographic information is included at the beginning of each typeset abstract.

As of June 19, 1992, the library consisted of 279 MIZAR articles authored by some 60 people. 214 theorem files that are referenced from other articles contained 5810 theorems; there were 109903 cross-references among articles. Totally there were about 15 MB of source text files with articles. Although the majority of articles have been authored by people from Białystok, many papers have been written by mathematicians from other universities in Poland, and there are articles written by foreign authors (Canada, Japan, Spain, USA).

The nature of the articles varies. Most of them are MIZAR translations of basic mathematics. Few of them contain new results. To get an idea about the contents of the library, look at Table 13 for a sample of article titles.

No.	Name	Inclusion date	Title	Author(s)
1	`BOOLE`	6.I.1989	*Boolean Properties of Sets*	Z. Trybulec and H. Święczkowska
17	`FUNCT_2`	6.IV.1989	*Functions from a Set to a Set*	Cz. Byliński
33	`WELLORD2`	26.IV.1989	*Zermelo Theorem and Axiom of Choice*	G. Bancerek
67	`FRAENKEL`	7.II.1990	*Function Domains and Fraenkel Operator*	A. Trybulec
106	`TRANSLAC`	12.VI.1990	*Translations in Affine Planes*	H. Oryszczyszyn and K. Prażmowski
185	`MEASURE1`	15.X.1990	*The σ-additive Measure Theory*	J. Białas
231	`ALI2`	17.VII.1991	*Fix Point Theorem for Compact Spaces*	A. de la Cruz
248	`HEINE`	21.XI.1991	*Heine-Borel's Covering Theorem*	A. Darmochwał and Y. Nakamura
274	`MIDSP_3`	28.V.1992	*Reper Algebras*	M. Muzalewski

Table 13: Sample of titles from MIZAR library.

The development of the MIZAR library may be perceived as an experiment in the sociology of mathematics. The acceptance criteria are very liberal: every submitted paper that is accepted by the MIZAR processor is included into the library. There are some efforts towards having automated reviewers (that looks, for example, for repeated theorems or trivial ones). There is a collection of programs called enhancers and improvers that try to automatically meliorate the submitted articles. The melioration makes changes in the submitted articles, that is, replaces a proof by a one step inference referencing a number of propositions, or replaces a sequence of inferences by a single one, or removes references to the superfluous premises in an inference step.

As the MIZAR system evolves, and this includes the input language, there is a need to rewrite pieces of the library articles to mirror the changes. To a large extent, this is done automatically. Sometimes, however, a manual intervention is required.

The people maintaining the library collect statistics about the references across articles. Consideration is given to whether or not the frequently quoted theorems or definition should be built into the MIZAR verifier. This was the fate of proposition `TARSKI:1` which stated that everything was a set, see Appendix A.

Table 14 contains the list of the 10 most frequently quoted theorems.

No. of ref.	% of all ref.	Name	Statement of the theorem
2347	2.1355%	`BOOLE:11`	x ∈ X & X ⊆ Y **implies** x ∈ Y
1642	1.4940%	`BOOLE:9`	x ∈ X∩Y **iff** x ∈ X & x ∈ Y
1558	1.4176%	`TARSKI:3`	X = {y} **iff for** x **holds** x ∈ X **iff** x = y
1349	1.2274%	`BOOLE:8`	x ∈ X∪Y **iff** x ∈ X **or** x ∈ Y
1252	1.1392%	`BOOLE:def 1`	Z = ∅ **iff not ex** x **st** x ∈ Z
1242	1.1301%	`BOOLE:29`	X ⊆ Y & Y ⊆ Z **implies** X ⊆ Z
964	0.8771%	`FINSEQ_1:13`	k = len p **iff** Seg k = dom p
922	0.8389%	`BOOLE:64`	(X∪Y)∪Z = X∪(Y∪Z)
848	0.7716%	`BOOLE:5`	X ⊆ Y **iff for** x **holds** x ∈ X **implies** x ∈ Y
792	0.7206%	`AXIOMS:2`	X **is** Subset **of** Y **iff** X ⊆ Y

Table 14: Top 10 theorems of MIZAR library.

8 The future

The MIZAR **language:** The logical level of the language has long been fixed. It is the type hierarchy that fuels all the changes. Recently, the development has focused on introducing a mechanism for deriving MIZAR structures in the spirit of the object-oriented approach, and on deriving new MIZAR modes by adding attributes to existing ones. Both proposed derivation techniques result in Boolean algebras of structures and sets of attributes, respectively. According to A. Trybulec both these changes will have a dramatic impact on the style of doing mathematics in MIZAR.

The language still lacks some polymorphic or generic facilities such that one has to prove analogous facts twice, for example, about lower and upper semilattices.

Translations: It is planned to implement the mechanical translation of MIZAR texts into other existing systems for doing mathematics, and vice versa. However, H. Barendregt's optimism on the time frame required for such a work is not commonly shared.

Large data base: A large data base would require a major effort from numerous parties and the administrative problems of such an enterprise should not be neglected. It is estimated that maintenance of a data base 10 times bigger than the current state (i.e. with about 3000 articles by several hundred authors) could stabilize a number of issues whose current solutions tend to be unstable.

Presentation: The usefulness of *Formalized Mathematics* containing typeset abstracts instigated some thoughts on typesetting entire MIZAR articles. Similarly, a need arises to develop some automated techniques for extracting topical monographs from the MIZAR library.

Accommodator: The process of preparing a local environment for checking a single article awaits a better solution. The problem here is similar to linking a newly written program with library modules, known to be a challenge in software engineering. An accommodator is expected to speed up the process of checking articles by cutting off the complex data base interaction at a certain level.

Neglected issues: There are some aspects of the MIZAR system that draw immediate criticism. To name a few: restriction to IBM PC and compatibles; poor user interface restricted to the text editor level; only textual searches of the data base; weak inference checker. All of them have been recognized as problems to work on but were perceived as second priority issues. Eventually, they have to be addressed.

Hibernation: Freezing the changes in the input language and in the MIZAR processor has been a goal for quite a while, yet it seems to move away like the horizon when you try to approach it.

9 How to learn Mizar?

The MIZAR language, its processor, and the organization of the MIZAR library evolve, and therefore there is not much in the way of written documentation, see [1].

In the face of documentation shortages the best way to learn MIZAR is to spend approximately four weeks in Białstok[6] and co-author a MIZAR article with a native user of the system. However, numerous cases are known of MIZAR users who that advanced their knowledge of the system by studying the existing texts (and there are 15MB of these).

Acknowledgements

I am indebted to all members of the MIZAR development group (which I was a member of years ago) and to all the authors of articles in MIZAR Library. Special thanks are to Andrzej Trybulec, Roman Matuszewski, Czesław Byliński, Edmund Woronowicz, Grzegorz Bancerek, and Zbigniew Karno.

This "commercial" has been written with the hope that their work meets with the recognition it deserves.

[6]Other places include: Łódź and Rzeszów in Poland, Madrid in Spain, and Nagano in Japan.

References

[1] Ewa Bonarska. *An Introduction to PC Mizar.* Mizar Users Group. Fondation Philippe le Hodey, Brussels, 1990.

[2] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988.

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

[4] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and Hindley J. R., editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[5] Andrzej Grzegorczyk. *Zarys arytmetyki teoretycznej.* PWN Warszawa, 1971.

[6] G. Huet and G. Plotkin, editors. *Proceedings of the 1st Workshop on Logical Frameworks.* ESPRIT BRA 3245, 1990. Anonymous ftp: `nuri.inria.fr`.

[7] G. Huet and G. Plotkin, editors. *Proceedings of the 2nd Workshop on Logical Frameworks.* ESPRIT BRA 3245, 1991. Anonymous ftp: `colonsay.dcs.ed.ac.uk`.

[8] S. Jaśkowski. On the rules of supposition in formal logic. *Studia Logica*, 1, 1934.

[9] H. C. Kennedy, editor. *Selected works of Giuseppe Peano.* University of Toronto Press, 1973.

[10] J. Ketonen. EKL—a mathematically oriented proof checker. In *Proceedings of 7th Int. Conf. on Automated Deduction*, pages 65–79, Napa, CA, May 1984.

[11] D. A. McAllester. *Ontic.* The MIT Press, 1989.

[12] S. McCall, editor. *Polish Logic in 1920–1939.* Clarendon Press, Oxford, 1967.

[13] M. Mostowski and Z. Trybulec. A certain experimental computer aided course of logic in Poland. In *Proceedings of World Conference on Computers in Education*, Norfolk, VA, 1985.

[14] Y. Nakamura. A language for description of mathematics—THEAX. Technical report, Shinshu University FIE, Nagano City, Japan, 1985. In Japanese.

[15] S. Nieva Soto. The Reasoner of MIZAR/LOG. *Computerized Logic Teaching Bulletin*, 2(1):22–35, March 1989.

[16] K. Ono. On a practical way of describing formal deductions. *Nagoya Mathematical Journal*, 21, 1962.

[17] Giuseppe Peano. *Opere Scelte.* Edizioni Cremonese, Roma, 1958.

[18] K. Prażmowski, P. Rudnicki, et al. Mizar-MSE Primer and User Guide. TR 88-9, The University of Alberta, Department of Computing Science, Edmonton, 1988.

[19] P. Rudnicki. Obvious inferences. *Journal of Automated reasoning*, 3:383–393, 1987.

[20] P. Rudnicki. What should be proved and tested symbolically in formal specifications? In *4th IEEE International Workshop on Software Specification and Design*, pages 190–195, Monterey, Ca., 1987.

[21] P. Rudnicki and W. Drabent. Proving properties of Pascal programs in MIZAR 2. *Acta Informatica*, 22:311–331, 1985. Erratum pp. 699–707.

[22] R.L. Smith et al. Computer-assisted axiomatic mathematics: Informal rigor. In O. Lecarme and R. Lewis, editors, *Computers in Education*, pages 803–809. North Holland, 1975.

[23] Alfred Tarski. Über unerreichbare Kardinalzahlen. *Fundamenta Mathematicae*, 30:68–89, 1938.

[24] Alfred Tarski. On well-ordered subsets of any set. *Fundamenta Mathematicae*, 32:176–183, 1939.

[25] A. Trybulec and H. Blair. Computer aided reasoning. In R. Parikh, editor, *Logic of Programs, LNCS 193*. Springer Verlag, 1985.

[26] A. Trybulec and H. Blair. Computer assisted reasoning with Mizar. In *Proceedings of the 9th IJCAI*, pages 26–28, Los Angeles, Ca., 1985.

[27] Andrzej Trybulec. Tarski Grothendieck Set Theory. *Formalized Mathematics*, 1:9–11, 1990.

[28] L. S. van Benthem Jutting. The development of a text in AUT-QE. In *Proceedings of APLASM'73, Symposium d'Orsay sur la Manipulation des Symboles at d'Utilisation d'APL*, Universite Paris XI, 1973.

A Tarski Grothendieck Set Theory

The following is the abstract and the actual article on which the MIZAR library is built. For obvious reasons the article has not been checked for validity. Unfortunately for the international audience the comments in text are mainly in Polish. First, the English summary provided by A. Trybulec (author).

> This is the first part of the axiomatics of the Mizar system. It includes the axioms of the Tarski Grothendieck set theory. They are: the axiom stating that everything is a set, the extensionality axiom, the definitional axiom of the singleton, the definitional axiom of the pair, the definitional axiom of the union of a family of sets, the definitional axiom of the boolean (the power set) of a set, the regularity axiom, the definitional axiom of the ordered pair, the Tarski's axiom A introduced in [23] (see also [24]), and the Frænkel scheme. Also, the definition of equinumerosity is introduced.

```
environ vocabulary EQUI_REL, BOOLE, FAM_OP;
  :: Andrzej Trybulec
  :: Teoria mnogosci Tarskiego Grothendiecka
begin
 reserve x,y,z,u for Any,
         N,M, X,Y,Z for set;
:: axiom  Tarski:1       ---- wszystko jest zbiorem
                    canceled; :: as obvious:  x is set;
axiom  :: Tarski:2       ---- ekstensjonalnosc zbiorow
                    (for x holds x ∈ X iff x ∈ Y) implies X = Y;
 :: Singletony i pary
                    definition
                      let y; func { y } -> set means
:: TARSKI: def 1
                       x ∈ it iff x = y;
                      let z; func { y, z } -> set means
:: TARSKI: def 2
                       x ∈ it iff x = y or x = z;
                    end;
axiom  :: Tarski:3       ---- definicja singletonu
                    X = { y } iff for x holds x ∈ X iff x = y;
axiom  :: Tarski:4       ---- definicja pary nieuporzadkowanej
                    X = { y,z } iff for x holds x ∈ X iff x = y or x = z;
                    definition let X,Y;
                     pred X c= Y means
:: TARSKI: def 3
                      x ∈ X implies x ∈ Y;
                      reflexivity;
                    end;
                    definition let X;
                       func union X -> set means
:: TARSKI: def 4
                        x ∈ it iff ex Y st x ∈ Y & Y ∈ X;
                    end;
axiom  :: Tarski:5       ---- definicja unii rodziny zbiorow
                    X = union Y iff for x holds x ∈ X iff ex Z st x ∈ Z & Z ∈ Y;
axiom  :: Tarski:6       ---- definicja zbioru potegowego
                    X = bool Y iff for Z holds Z ∈ X iff Z c= Y;
```

```
axiom  :: Tarski:7        ---- aksjomat regularnosci
                   x ∈ X implies ex Y st Y ∈ X & not ex x st x ∈ X & x ∈ Y;

 scheme Fraenkel { A()-> set, P[Any, Any] }:
  ex X st for x holds x ∈ X iff ex y st y ∈ A() & P[y,x]
  provided for x,y,z st P[x,y] & P[x,z] holds y = z;

                   definition let x,y;
                    func [x,y] means
:: TARSKI: def 5
                    it = { { x,y }, { x } };
                   end;
axiom  :: Tarski:8        ---- definicja pary uporzadkowanej
                   [ x,y ] = { { x,y }, { x } };
                   definition let X,Y;
                    pred X ≈ Y means
:: TARSKI: def 6
                     ex Z st
                           (for x st x ∈ X ex y st y ∈ Y & [x,y] ∈ Z) &
                           (for y st y ∈ Y ex x st x ∈ X & [x,y] ∈ Z) &
                           for x,y,z,u st [x,y] ∈ Z & [z,u] ∈ Z holds x = z iff y = u;
                   end;
:: Alfred Tarski
:: Ueber unerreichbare Kardinalzahlen,
:: Fundamenta Mathematicae, vol.30 (1938), pp.68-69
:: Axiom A. (Axiom der unerreichbaren Mengen). Zu jeder Menge N gibt es
:: eine Menge M mit folgenden Eigenschaften :
:: A1. N ∈ M;
:: A2. ist X ∈ M und Y c= X, so ist Y ∈ M;
:: A3. ist X ∈ M und ist Z die Menge, die alle Mengen Y c= X und keine
::     andere Dinge als Element enthaelt, so,ist z ∈ M;
:: A4. ist X c= M und sind dabei die Menge X und M nicht gleichmaechtig,
::     so ist X ∈ M.
:: takze
:: Alfred Tarski
:: On Well-ordered Subsets of any Set,
:: Fundamenta Mathematicae, vol.32 (1939), pp.176-183
:: A. For every set N there exists a system M of sets which satisfies
::    the following conditions :
:: (i)    N ∈ M
:: (ii)   if X ∈ M and Y c= X, then Y ∈ M
:: (iii)  if X ∈ M and Z is the system of all subsets of X, then Z ∈ M
:: (iv)   if X c= M and X and M do not have the same potency, then X ∈ M.

axiom :: Tarski:9
                     ex M st N ∈ M &
                       (for X,Y holds X ∈ M & Y c= X implies Y ∈ M) &
                       (for X holds X ∈ M implies bool X ∈ M) &
                       (for X holds X c= M implies X ≈ M or X ∈ M);
```

B Built-in Concepts

The English summary provided by A. Trybulec (author):

> This abstract contains the second part of the axiomatics of the Mizar system (the first part is in abstract [27]). The axioms listed here characterize the Mizar built-in concepts that are automatically attached to every Mizar article. We give definitional axioms of the following concepts: element, subset, Cartesian product, domain (non-empty subset), subdomain (non empty-subset of a domain), set domain (domain consisting of sets). Axioms of strong arithmetics of real numbers are also included.

Numerous axioms that were needed some time ago have been cancelled; they have been built into the processor and they are now obvious to the verifier. The trace of them is required to properly process older articles that made references to the axioms.

Polish comments have been deleted from the text.

```
 :: Andrzej Trybulec

environ
 vocabulary Boole;    signature Tarski;
begin
  reserve x,y,z for Any,
          X,X1,X2,X3,X4,Y for set;
:: axiom AXIOMS:1  (ex x st x ∈ X) implies (x is Element of X iff x ∈ X);
                  canceled; :: as obvious
axiom :: AXIOMS:2
                  X is Subset of Y iff X c= Y;
axiom :: AXIOMS:3
                  z ∈ [:X,Y:] iff ex x,y st x ∈ X & y ∈ Y & z = [x,y];
axiom :: AXIOMS:4
                  X is non-empty implies ex x st x ∈ X;
axiom :: AXIOMS:5
                  [: X1,X2,X3 :] = [:[:X1,X2:],X3:];
axiom :: AXIOMS:6
                  [: X1,X2,X3,X4 :] = [:[:X1,X2,X3:],X4:];
 reserve D1,D2,D3,D4 for non-empty set;
:: axiom AXIOMS:7  for X being Element of [: D1,D2 :] holds X is TUPLE of D1, D2;
                  canceled; :: as obvious
:: axiom AXIOMS:8  for X being Element of [: D1,D2,D3 :] holds X is TUPLE of D1, D2, D3;
                  canceled; :: as obvious
:: axiom AXIOMS:9  for X being Element of [: D1,D2,D3,D4 :] holds X is TUPLE of D1, D2, D3, D4;
                  canceled; :: as obvious
 reserve D for non-empty set;
axiom :: AXIOMS:10
                  D1 is non-empty Subset of D2 iff D1 c= D2;
:: axiom AXIOMS:11  D is SET DOMAIN;
                  canceled; :: as obvious
 reserve x,y,z for Element of REAL;
axiom :: AXIOMS:12
                  x + y = y + x;
axiom :: AXIOMS:13
                  x + (y + z) = (x + y) + z;
axiom :: AXIOMS:14
                  x + 0 = x;
axiom :: AXIOMS:15
```

```
                 x · y = y · x;
axiom :: AXIOMS:16
                 x · (y · z) = (x · y) · z;
axiom :: AXIOMS:17
                 x · 1 = x;
axiom :: AXIOMS:18
                 x · (y + z) = x · y + x · z;
axiom :: AXIOMS:19
                 ex y st x + y = 0;
axiom :: AXIOMS:20
                 x <> 0 implies ex y st x · y = 1;
axiom :: AXIOMS:21
                 x ≤ y & y ≤ x implies x = y;
axiom :: AXIOMS:22
                 x ≤ y & y ≤ z implies x ≤ z;
axiom :: AXIOMS:23
                 x ≤ y or y ≤ x;
axiom :: AXIOMS:24
                 x ≤ y implies x + z ≤ y + z;
axiom :: AXIOMS:25
                 x ≤ y & 0 ≤ z implies x · z ≤ y · z;
axiom :: AXIOMS:26
                 for X,Y being Subset of REAL st
                  (ex x st x ∈ X) & (ex x st x ∈ Y) &
                   for x,y st x ∈ X & y ∈ Y holds x ≤ y
                  ex z st
                   for x,y st x ∈ X & y ∈ Y holds x ≤ z & z ≤ y;
:: axiom AXIOMS:27  x is Real;
                 canceled; :: as obvious
axiom :: AXIOMS:28
                 x ∈ NAT implies x + 1 ∈ NAT;
axiom :: AXIOMS:29
                 for A being set of Real
                   st 0 ∈ A & for x st x ∈ A holds x + 1 ∈ A holds NAT c= A;
 reserve i,j,k for Nat;
axiom :: AXIOMS:30
                k = { i: i<k };
```

C A Mizar article

The following is the text of an article from the Main Mizar Library. This article is unusual—it is shortest in the library. In order to decrease the number of text lines and with hope of improving readability I have manipulated the white space of the original submission. The characters of extended ASCII have been replaced by some symbols available in LaTeX. The MIZAR processor restricts the length of input lines to 80 characters. The text below violates this restriction for presentation purposes.

```
:: Alicia de la Cruz
:: Fix Point Theorem for Compact Spaces

environ
 vocabulary METRYKA, SFAMILY, POWER1, FINITE, SEQ1, SEQ2, SEQM, SUB_OP, REAL_1,
            TOPCON, PCOMPS, FUNC, TOP2, FAM_OP, BOOLE, FINITER2, ALI2, FUNC_REL;
 signature FINSET_1, METRIC_1, FUNCT_1, FUNCT_2, PRE_TOPC, POWER, BOOLE, FUNCOP_1,
           TARSKI, COMPTS_1, PCOMPS_1, SETFAM_1, TOPS_1, TOPS_2, SEQ_1, SEQ_2, SEQM_3,
           SUBSET_1, REAL_1, NAT_1;
 definitions COMPTS_1, TARSKI, TOPS_2, FUNCT_2;
 theorems METRIC_1, SUBSET_1, REAL_1, PCOMPS_1, REAL_2, COMPTS_1, POWER, BOOLE, SEQ_2,
          SEQ_4, SERIES_1, SEQM_3, AXIOMS, SETFAM_1, TARSKI, SEQ_1, PRE_TOPC, TOPS_1,
          SQUARE_1, DOMAIN_1, FUNCOP_1, ZFMISC_1;
 schemes SETFAM_1, SEQ_1, GROUP_4, FINSET_1, NAT_1;

begin
      reserve M for MetrSpace, x, y for Element of the carrier of M;
theorem VIT:
       for F being set st F is_finite & F <> ∅ &
                         for B, C being set st B ∈ F & C ∈ F holds B c= C or C c= B
             ex m being set st m ∈ F & for C being set st C ∈ F holds m c= C
 proof    pred P[set] means
          $1 <> ∅ implies ex m being set st m ∈ $1 & for C being set st C ∈ $1 holds m c= C;
  let F be set such that
   Z: F is_finite   and    Y: F <> ∅ and
   X: for B, C being set st B ∈ F & C ∈ F holds B c= C or C c= B;
   A: P[∅];
   B: now let x, B be set such that j0: x ∈ F & B c= F & P[B];
         now per cases;                          :: we have to prove P[B U{x}]
          case j: not ex y being set st y ∈ B & y c=x;
           assume B U {x} <> ∅;   take m = x;   x ∈ {x} by TARSKI:def 1;
           hence m ∈ B U {x} by BOOLE:8;   let C be set;   assume C ∈ B U {x};   then
            _1_: C∈B or C∈{x} by BOOLE:8;   then j1:C∈B or C=x by TARSKI:def 1;
            j2: not C c=x or C=x by j, TARSKI:def 1, _1_;   C ∈ F by j0, BOOLE:11, j1;
           hence m c= C by j0, X, j2;
          case ex y being set st y ∈ B & y c=x;   then consider y being set such that
            j5: y ∈ B & y c=x;   assume B U {x} <> ∅;   consider m being set such that
            j3:m ∈ B   and
            j4: for C being set st C ∈ B holds m c= C by j0, j5, BOOLE:def 1;
                m c= y by j4, j5;   then j6:m c= x by j5, BOOLE:29;
           take m;   thus m ∈ B U {x} by j3, BOOLE:8;
           let C be set;   assume C ∈ B U {x};   then C ∈ B or C ∈ {x} by BOOLE:8;
           hence m c= C by j4, j6, TARSKI:def 1;            end;
         hence P[B U {x}];        end;
     P[F] from Finite(Z, A, B);
  hence thesis by Y;  end;
```

```
definition   let M be MetrSpace;
 mode contraction of M -> Function of  the carrier of M, the carrier of M
         means :DD: ex L being Real st 0<L & L<1 &
                        for x, y being Point of M holds dist(it.x, it.y)≤L·dist(x, y);
 existence   proof   consider x being Point of M;
    (the carrier of M) --> x is Function of the carrier of M, the carrier of M   proof
     thus dom ((the carrier of M) --> x) = the carrier of M by FUNCOP_1:19;
        cc:rng ((the carrier of M) --> x) c= {x} by FUNCOP_1:19;
        {x} c= the carrier of M by ZFMISC_1:37;
     hence rng ((the carrier of M) --> x) c= the carrier of M by BOOLE:29, cc;     end;
    then reconsider f = (the carrier of M) --> x
                   as  Function of  the carrier of M, the carrier of M;
  take f,  1/2; 0<1;   hence aa: 0<1/2 & 1/2<1 by SEQ_2:3, SQUARE_1:3;
  let z, y being Point of M;   f.z=x & f.y=x by FUNCOP_1:13;
     then bb: dist(f.z, f.y) = 0 by METRIC_1:def 3;   dist(z, y)≥0 by METRIC_1:7;
  hence dist(f.z, f.y)≤(1/2)·dist(z, y) by bb, aa, REAL_2:121;      end;
end;

theorem   for f being contraction of M st TopSpaceMetr(M) is_compact
             ex c being Point of M st f.c =c &            :: exists a fix point
             for x being Point of M st f.x=x holds x=c    :: exactly 1 fix point
proof
 let f be contraction of M;   consider L being Real such that
  a1: 0<L & L<1 and
  a2: for x, y being Point of M holds dist(f.x, f.y)≤L·dist(x, y) by DD;
 assume a7: TopSpaceMetr(M) is_compact;   consider x0 being Point of M;   set a=dist(x0, f.x0);
  now assume a <> 0;
        consider F being Subset-Family of the carrier of TopSpaceMetr(M) such that
    kkk: for B being Subset of the carrier of TopSpaceMetr(M) holds B∈F iff
             ex n being Nat st B = { x where x is Point of M : dist(x, f.x) ≤ a·L to_power n }
                                          from SubFamEx;
    TopSpaceMetr(M) = TopStruct≪the carrier of M, Family_open_set(M)≫ by PCOMPS_1:def 6; then
    d5: the carrier of M = the carrier of TopSpaceMetr(M);
        set B = { x where x is Point of M : dist(x, f.x) ≤ a·L to_power (0+1) };
        B is Subset of the carrier of M from SubsetD;   then B ∈ F by kkk, d5;   then
    d1: F<>∅ by BOOLE:def 1;
    a8: F is_centered   proof   thus F <> ∅ by d1;
     let G be Subset-Family of the carrier of TopSpaceMetr(M) such that
      b1: G <> ∅   and   b2: G c= F   and   b3: G is_finite;
      for B, C being set st B ∈ G & C ∈ G holds B c= C or C c= B   proof
       let B, C be set ;   assume h0:B ∈ G & C ∈ G;   then h3: B ∈ F & C ∈ F by b2, BOOLE:11;
            B is Subset of the carrier of TopSpaceMetr(M) by SETFAM_1:43, h0;
            then consider n being Nat such that
        h4: B = { x where x is Point of M : dist(x, f.x) ≤ a·L to_power n } by kkk, h3;
            C is Subset of the carrier of TopSpaceMetr(M) by SETFAM_1:43, h0;
            then consider m being Nat such that
        h5: C = { x where x is Point of M : dist(x, f.x) ≤ a·L to_power m } by kkk, h3;
        lemma1: for n, m being Nat st n≤m holds L to_power m ≤ L to_power n   proof
         let n, m being Nat such that iii: n≤m;
          now per cases by iii, REAL_1:def 5;
           case n<m;   then L to_power n > L to_power m by POWER:45, a1;
            hence L to_power n ≥ L to_power m by REAL_1:def 5;
           case n=m;   hence L to_power n ≥ L to_power m;     end;
         hence thesis;          end;
        lemma2: for n, m being Nat st n≤m holds a·L to_power m ≤ a·L to_power n   proof
         let n, m being Nat such that iii: n≤m;
          now per cases;
           case c:  a=0; then a·L to_power m = 0 by REAL_1:20 .= a·L to_power n by REAL_1:20, c;
```

```
       hence a·L to_power m ≤ a·L to_power n;
      case a<>0;  cc:  a≥0 by METRIC_1:7;  L to_power m ≤ L to_power n by iii, lemma1;
       hence a·L to_power m ≤ a·L to_power n by cc, REAL_1:51;     end;
    hence thesis;         end;
   now per cases by AXIOMS:23;
    case i:n≤m;   thus C c= B   proof   let y be Any;   assume y ∈ C;
         then consider x being Point of M  such that
      C1: y = x   and   C2: dist(x, f.x) ≤ a·L to_power m by h5;
         a·L to_power m ≤ a·L to_power n by i, lemma2;    then
         dist(x, f.x) ≤ a·L to_power n by AXIOMS:22, C2;
      hence y ∈ B by C1, h4;   end;
    case ii:m≤n;   thus B c= C   proof   let y be Any;   assume y ∈ B;
         then consider x being Point of M  such that
      C1: y = x   and   C2: dist(x, f.x) ≤ a·L to_power n by h4;
         a·L to_power n ≤ a·L to_power m by ii, lemma2;   then
         dist(x, f.x) ≤ a·L to_power m by AXIOMS:22, C2;
      hence y ∈ C by C1, h5;   end;         end;
  hence B c= C or C c= B;       end;     then consider  m being set such that
 n1: m ∈ G   and   n2: for C being set st C ∈ G holds m c= C by VIT, b1, b3;
 n3: m c= meet G by SETFAM_1:6, b1, n2;   n4: m ∈ F by n1, BOOLE:11, b2;
 h5: m is Subset of the carrier of TopSpaceMetr(M) by SETFAM_1:43, n1;
 CC: L <> 0 by a1;   dist(x0,f.x0) = a·1 by REAL_1:7 .= a·L to_power 0 by POWER:29, CC;
     then x0 ∈ { x where x is Point of M : dist(x, f.x) ≤ a·L to_power 0 };   then
 P: { x where x is Point of M : dist(x,f.x) ≤ a·L to_power 0 } <> ∅ by BOOLE:def 1;
 P': for k being Nat st {x where x is Point of M : dist(x, f.x) ≤ a·L to_power k} <> ∅
                  holds {x where x is Point of M : dist(x,f.x) ≤ a·L to_power (k+1)} <> ∅
 proof let k be Nat; assume {x where x is Point of M : dist(x,f.x) ≤ a·L to_power k} <> ∅;
       then consider z being Any such that
   s1: z∈{ x where x is Point of M : dist(x, f.x) ≤ a·L to_power k } by BOOLE:def 1;
       consider y being Point of M such that y=z and s2: dist(y,f.y)≤ a·L to_power k by s1;
   L ≥ 0 by a1, REAL_1:def 5; then s4: L·dist(y,f.y) ≤ L·(a·L to_power k) by REAL_1:51, s2;
   s3: L·(a·L to_power k)  = L·a·L to_power k by AXIOMS:16  .= a·L·L to_power k by AXIOMS:15
        .= a·(L·L to_power k) by AXIOMS:16  .= a·(L to_power 1·L to_power k) by POWER:30
        .= a·(L to_power k·L to_power 1) by AXIOMS:15  .= a·L to_power (k+1) by POWER:32, a1;
       dist(f.y, f.(f.y)) ≤ L·dist(y, f.y) by a2;   then
       dist(f.y, f.(f.y)) ≤ a·L to_power (k+1) by s3, s4, AXIOMS:22;   then
       f.y ∈ { x where x is Point of M : dist(x, f.x) ≤ a·L to_power (k+1) };
  hence { x where x is Point of M : dist(x,f.x)≤a·L to_power (k+1) } <> ∅ by BOOLE:def 1;
 end;
 for k being Nat holds { x where x is Point of M : dist(x,f.x) ≤ a·L to_power k} <> ∅
                              from Ind(P, P');   then  m <> ∅ by h5, kkk, n4;
hence  meet G <> ∅ by BOOLE:30, n3;         end;
F is_closed   proof   let B being Subset of the carrier of TopSpaceMetr(M);
assume B ∈ F;   then consider n being Nat such that
 mm:  B= { x where x is Point of M : dist(x, f.x) ≤ a·L to_power n } by kkk;
 A1: TopSpaceMetr(M)=TopStruct≪the carrier of M, Family_open_set(M)≫ by PCOMPS_1: def 6;
     then reconsider V = B` as Subset of the carrier of M;    set B' = B`;
 for x being Point of M st x ∈ V ex r being Real st r>0 & Ball(x, r) c= V   proof
  let x be Point of M such that m1: x ∈ V;   take r = (dist(x, f.x)-a·L to_power n)/2;
       2<>0;   then 2·r = dist(x, f.x)-a·L to_power n by REAL_2:73;   then
   m20: dist(x,f.x)-2·r = a·L to_power n by REAL_2:12; not x∈B by m1, SUBSET_1:53; then
       dist(x,f.x)>a·L to_power n by mm; then dist(x,f.x)-a·L to_power n>0 by REAL_2:108;
  hence r>0 by SEQ_2:3;
  let z be Any;   assume m: z ∈ Ball(x, r);   Ball(x, r) c= the carrier of M by AXIOMS:2;
   then reconsider y=z as Point of M by BOOLE:11, m;   m8: dist(x, y)<r by METRIC_1:19, m;
        dist(x, y) + dist(y, f.y) ≥ dist(x, f.y) by METRIC_1:6;   then
   m13: (dist(x,y)+dist(y,f.y))+dist(f.y,f.x) ≥ dist(x,f.y)+dist(f.y,f.x) by REAL_1:53;
        dist(x, f.y) + dist(f.y, f.x) ≥ dist(x, f.x) by METRIC_1:6;   then
```

```
            dist(x, y)+dist(y, f.y)+dist(f.y, f.x)≥dist(x, f.x) by m13, AXIOMS:22;   then
            dist(y, f.y)+dist(x, y)+dist(f.y, f.x)≥dist(x, f.x) by AXIOMS:12;   then
       m6:  dist(y, f.y)+(dist(x, y)+dist(f.y, f.x))≥dist(x, f.x) by AXIOMS:13;
       m10: dist(x, y) = dist(y, x) by METRIC_1:5;   m3: dist(f.y, f.x)≤L·dist(y, x) by a2;
            dist(y, x)≥0 by METRIC_1:7;   then L·dist(y, x)≤dist(y, x) by REAL_2:147, a1;
            then dist(f.y, f.x)≤dist(y, x) by m3, AXIOMS:22;   then
       m4:  dist(f.y, f.x)+dist(y, x) ≤ dist(y, x)+dist(y, x) by REAL_1:49;
            2>0;   then 2·dist(x, y)<2·r by m8, REAL_1:70;   then
       m9:  dist(y, f.y) + 2·dist(x, y)< dist(y, f.y) + 2·r by REAL_1:59;
            dist(f.y, f.x)+dist(y, x) ≤ 2·dist(y, x) by m4, SQUARE_1:5;   then
            dist(y, x) + dist(f.y, f.x) ≤ 2·dist(y, x) by AXIOMS:12;   then
            dist(y,f.y)+(dist(y,x)+dist(f.y,f.x))≤dist(y,f.y)+2·dist(y,x) by REAL_2:103;
            then dist(y, f.y)+2·dist(x, y)≥dist(x, f.x) by m6, AXIOMS:22, m10;
            then dist(y, f.y)+2·r>dist(x, f.x) by REAL_1:58, m9;   then
            not ex x being Point of M st y=x & dist(x,f.x)≤a·L to_power n by m20, REAL_1:91;
           then m22: not y ∈ B by mm;   the carrier of M <> ∅ by DOMAIN_1:1;
      hence z ∈ V by SUBSET_1:50, m22, A1;       end;   then
     B‘ ∈ Family_open_set(M) by PCOMPS_1:def 5;   then B’ is_open by A1,  PRE_TOPC:def 4;
    hence B is_closed by TOPS_1:29;
   end;   then
   meet F<>∅ by COMPTS_1:13,a8,a7; then consider c’ being Point of TopSpaceMetr(M) such that
   d2: c’∈meet F by SUBSET_1:10;
       reconsider c = c’ as Point of M by d5;   consider s’ being Real_Sequence such that
   b3: for n being Nat holds s’.n=L to_power (n+1) from ExRealSeq;    set s = a □ s’;
   b6’’: s’ is_convergent & lim s’=0 by a1, SERIES_1:1, b3;   then
   b6: s is_convergent by SEQ_2:21;
   b6’: lim s = a·0  by b6’’, SEQ_2:22 .= 0 by REAL_1:20;
       consider r being Real_Sequence such that
   b4: for n being Nat holds r.n=dist(c, f.c) from ExRealSeq;
   b5: r is_constant by SEQM_3: def 5, b4; then b7: r is_convergent by SEQ_4:39;
   now let n be Nat;   set B = { x where x is Point of M : dist(x, f.x) ≤ a·L to_power (n+1)};
     B is Subset of the carrier of M from SubsetD;   then B ∈ F by kkk, d5;   then
     c∈ B by d1, SETFAM_1:1, d2;   then
     d3: ex x being Point of M st c = x & dist(x, f.x) ≤ a·L to_power (n+1);
     d3’: r.n = dist(c, f.c) by b4;   s.n = a·s’.n by SEQ_1:def 5 .= a·L to_power (n+1) by b3;
    hence r.n ≤ s.n by d3, d3’;       end;   then
   c1: lim r ≤ lim s by SEQ_2:32, b6, b7;   r.0=dist(c, f.c) by b4;   then
       dist(c, f.c)≤0 & dist(c, f.c)≥0 by c1, b6’, METRIC_1:7, SEQ_4:40, b5;   then
       dist(c, f.c)=0 by AXIOMS:21;
  hence ex c being Point of M st dist(c, f.c) = 0;
 end;   then consider c being Point of M such that
 XX: dist(c, f.c) = 0;
take c;
thus a4: f.c =c by METRIC_1: def 3, XX;
let x be Point of M ; assume a3: f.x=x; assume x<>c; then a6: dist(x,c)<>0 by METRIC_1:def 3;
 dist(x, c)≥0 by METRIC_1:7;   then dist(x, c)>0 by REAL_1:def 5, a6;   then
 L·dist(x, c)<dist(x, c) by a1, REAL_2:145;
hence contradiction by a3, a4, a2;
end;
```

D A Mizar abstract

Here is the abstract of the article presented in Appendix C. The abstracts are extracted mechanically from the submitted articles and the reference identification of MIZAR items is automatically inserted. If one wants to make a reference to another article, they have to use reference names as presented in the article's abstract.

```
:: Alicia de la Cruz
:: Fix Point Theorem for Compact Spaces

environ

 vocabulary METRYKA,SFAMILY,POWER1,FINITE,SEQ1,SEQ2,SEQM,SUB_OP,REAL_1,TOPCON,
     PCOMPS,FUNC,TOP2,FAM_OP,BOOLE,FINITER2,ALI2,FUNC_REL;
 signature FINSET_1,METRIC_1,FUNCT_1,FUNCT_2,PRE_TOPC,POWER,BOOLE,FUNCOP_1,
     TARSKI,COMPTS_1,PCOMPS_1,SETFAM_1,TOPS_1,TOPS_2,SEQ_1,SEQ_2,SEQM_3,
     SUBSET_1,REAL_1,NAT_1;

begin
  reserve M for MetrSpace,x,y for Element of the carrier of M;

theorem :: ALI2:1
for F being set st F is_finite & F <> ∅ &
 for B,C being set st B ∈ F & C ∈ F holds B c= C or C c= B
 ex m being set st m ∈ F & for C being set st C ∈ F holds m c= C;

definition  let M be MetrSpace;
mode contraction of M -> Function of  the carrier of M , the carrier of M
means :: ALI2:def 1
ex L being Real st 0<L & L<1& for x,y being Point of M holds
dist(it.x,it.y)≤L·dist(x,y);
end;

theorem :: ALI2:2
for f being contraction of M st TopSpaceMetr(M) is_compact
ex c being Point of M st f.c =c &            :: exists a fix point
for x being Point of M st f.x=x holds x=c;
```

E A TEXed Mizar abstract

MIZAR abstracts are further mechanically processed, typeset using TEX, and published (see footnote on page 296). Each author of an article accepted to the library must provide a title, the author's name and address, and a summary (in English) that are included into the typeset abstract.

The next two pages contain a copy of the typeset version of the abstract from Appendix D One can consider the English of the text rather poor, but it should be remembered that this text has been generated mechanically. The process of mechanical translation from MIZAR into English is still being worked on.

Kripke Semantics for a Logical Framework

Alex K. Simpson*
als@dcs.ed.ac.uk
Department of Computer Science, University of Edinburgh,
JCMB, The King's Buildings, Edinburgh, EH9 3JZ

July 1992

Abstract

We present a semantic analysis (using Kripke lambda models) of a meta-logic (minimal implicational predicate logic with quantification over all higher types) with emphasis on its use as a logical framework. An object-logic is encoded by a meta-theory axiomatising its consequence relation. Properties of the encoding, such as adequacy, are analysed in terms of corresponding semantic properties. In the case of adequacy we are able to prove the interesting "faithfulness" direction by showing that the meta-theory has a certain kind of model. The semantics is also used to analyse conditions under which the admissibility of an inference rule is provable in the meta-logic.

1 Introduction

The aim of this paper is to give a semantic analysis of the use of a meta-logic as a logical framework.

The meta-logic we consider is a fragment of intuitionistic predicate logic with quantification over all higher types. The idea of using such a logic as a framework goes back to the Edinburgh Logical Framework (LF) [6] (although this was itself partly inspired by de Bruijn's AUTOMATH [3] and Martin-Löf's system of arities [11]). Although LF is a dependently-typed lambda-calculus, it can be viewed as a fragment of intuitionistic logic using the propositions as types correspondence. Intuitively, however, it is quite natural to separate types from propositions. For example, the LF type (we follow the notation of [6]):

$$\Pi\varphi\!:\!o.\Pi\psi\!:\!o.true(\varphi) \to true(\varphi \vee \psi) \tag{1}$$

can be read naturally as the proposition:

$$\forall\varphi\!:\!o.\ \forall\psi\!:\!o.\ true(\varphi) \supset true(\varphi \vee \psi) \tag{2}$$

One is thus replacing the "judgements as types" principle of [6] with some kind of meta-axiomatisation. This separation between propositions and types has been made explicit in the applications of Isabelle and λProlog as logical frameworks (see [12, 4]).

There are a number of advantages to the LF approach. One merit is the obvious simplicity of having a calculus with only one connective (dependent product). Another is that the judgements

*Supported by SERC grant no. 90311820.

as types principle enables proofs to be explicitly represented as terms of the framework. With this understanding the type (1) is the natural type for the $\vee$-introduction rule which, given two formulae φ and ψ together with a proof of φ, constructs a proof of $\varphi \vee \psi$. Finally, the idea of separating propositions and types, though natural in some cases, might be less so in general. The single ontology of LF allows for the maximum flexibility in making encodings.

Nevertheless there are also good reasons for separating propositions and types. One is that a framework should distinguish between syntax and logical consequence. Syntax will still be represented by terms of appropriate type, but now consequence is expressed by a meta-axiomatisation. However, these kind of separations can also be made to good effect in a wholly type-theoretical setting (see [5]). A second argument is to prefer the meta-logical approach on the grounds that the main benefit offered by the type-theoretic approach, explicit proof representation, is neither important nor desirable. Proof representation is unimportant if one takes the point of view that the feature of a logic of real interest is its consequence relation (see section 4). An argument for the undesirability of proof representation, based on efficiency, is given in [12]. A further argument against the kind of proof representation offered by LF is that it is too crude to capture many interesting proof systems (two sided sequent calculi for example). In fact the numerous examples in [1] testify to LF as being good for representing consequence in the general case, but good for representing proofs only in particularly well-behaved cases.

However, the main reason we are interested in distinguishing propositions and types is that we wish to consider a logical framework as a meta-logic. We want to understand the encoding of the logic as a commitment to certain meta-propositions about the logic (the proposition (2) for example). In order to understand these meta-propositions mathematically, we require a semantics for the meta-logic. Moreover, we want to relate the meta-propositions to the encoded logic. So the semantics should be able to provide links between the encoded logic and its meta-theory. These links will turn out to be very concrete. For example, in section 5 we build a model of a meta-theory out of models of the logic it encodes.

The structure of this paper is as follows. In section 2 we introduce the meta-logic. In section 3 we give it a Kripke semantics, proving soundness and completeness. Section 4 is then a necessary preliminary to the rest of the paper. In it we survey various elementary properties of consequence relations. The bulk of the paper is contained sections 5 and 6. In the former we consider how the meta-logic can be used to encode logics. The property we are most interested in is the "adequacy" of an encoding. This is given a semantic characterisation, which is then used to give an example proof of adequacy. Finally, in section 6 we consider a notion of admissible rule for a logic. The semantics is used to analyse under what conditions the admissibility of a rule is provable in the meta-theory representing a logic.

Throughout this paper we consider many different sequent-style judgements with sets for the antecedent. In these we adopt the standard notational conventions of writing "," for "$\cup$", omitting set delimiters from singleton sets and omitting explicit mention of the empty set.

2 The meta-logic

The meta-logic is minimal implicational predicate logic with universal quantification over all higher types. It is quite similar to the logics considered in [12, 4].

We use A, B and C to range over (simple) types, M and N to range over terms (of the simply-typed λ-calculus), and Φ and Ψ to range over formulae (lower case Greek letters will be reserved for formulae of the object-logic).

We assume given four countably infinite, disjoint sets: a set of *type constants*, a set of *predicate symbols*, a set of *term constants* and a set of *variables*. We use P to range over the predicate symbols, c to range over the term constants and x to range over the variables.

A theory is generated by a *presentation* which is a quadruple, $(\mathcal{T}, \mathcal{P}, \Sigma, \mathcal{A})$, where each of $\mathcal{T}$, $\mathcal{P}$, Σ and $\mathcal{A}$ are sets as specified below. Mostly (but not exclusively) we consider *finite* presentations, ie. those in which all four sets are finite.

$\mathcal{T}$ is a subset of the set of type constants. Types are generated from this set by the grammar:

$$A ::= \alpha \mid A \to B$$

where α ranges over elements of $\mathcal{T}$. As usual, when brackets are omitted, "$\to$" associates to the right. $\mathcal{P}$ is a set of *predicate declarations* of the form $P : \langle A_1, \ldots, A_n \rangle$ (where n is possibly zero) such that each predicate symbol, P, appears only once in the set. Σ is a set of *constant declarations* of the form $c : A$ such that each term constant, c, appears only once in the set. The requirements on $\mathcal{A}$ are given below. Henceforth everything will be parameterised over $\mathcal{T}$ and $\mathcal{P}$ and these sets will usually be left implicit. Thus we often refer to the presentation as $(\Sigma, \mathcal{A})$.

A *context*, Γ, is a finite set of *variable declarations* of the form $x : A$ such that each variable, x, appears only once in the set. The abstract syntax of terms and formulae is given by the following grammar.

$$\begin{array}{rcl} M & ::= & c \mid x \mid \lambda x : A.\, M \mid M(N) \\ \Phi & ::= & P(M_1, \ldots, M_n) \mid \Phi \supset \Psi \mid \forall x : A.\, \Phi \end{array}$$

We assume that the reader understands the notion of free and bound variable. We write $N[M/x]$ and $\Phi[M/x]$ for the substitution of M for all free occurrences of x in N and Φ respectively. Lambda-terms and quantified formulae are considered identified up to α-equivalence.

The term calculus is just the simply-typed lambda calculus (for which a good reference is [9]). We write $\Gamma \rhd_\Sigma M : A$ to mean that M is term over Σ with type A in context Γ. We shall only be concerned with $\beta\eta$-equality, $=_{\beta\eta}$, between terms. We note (but shall not use) that equality between terms is decidable. A term, M, such that $\Gamma \rhd_\Sigma M : A$ is said to be in *long-$\beta\eta$ normal form* (with respect to Γ and Σ) if it has the form:

$$\lambda x_1 : A_1.\, \ldots \lambda x_n : A_n.\, h(M_1) \ldots (M_m)$$

where: $n, m \geq 0$; h is either a variable or a constant; $\Gamma, x_1 : A_1, \ldots, x_n : A_n \rhd_\Sigma h(M_1) \ldots (M_m) : \alpha$ for some type constant α; and each M_i $(1 \leq i \leq m)$ is in long-$\beta\eta$ normal form with respect to $\Gamma, x_1 : A_1, \ldots, x_n : A_n$ and Σ. Clearly any M in long-$\beta\eta$ normal form with respect to Γ and Σ is also in long-$\beta\eta$ normal form with respect to $\Gamma' \supseteq \Gamma$ and $\Sigma' \supseteq \Sigma$. The crucial property of long-$\beta\eta$ normal forms is the following (see [8]): if $\Gamma \rhd_\Sigma M : A$ then there is a unique term, $\beta\eta(M)$, in long-$\beta\eta$ normal form (with respect to Γ and Σ) such that $M =_{\beta\eta} \beta\eta(M)$.

In Figure 5 we give a formal system for deriving judgements of the form $\Gamma \rhd_\Sigma \Phi$ prop. We often write simply $\Gamma \rhd_\Sigma \Phi$ prop to mean that the given judgement is derivable, in which case we say that Φ is *well-formed* in Γ and Σ. When Γ is the empty set it is omitted from such statements. Σ will only be omitted when it can be understood from the context.

The fourth component of the presentation, $\mathcal{A}$, is a set of formulae, the *axioms*, such that each formula in $\mathcal{A}$ is well-formed in Σ.

Logical consequence for $(\Sigma, \mathcal{A})$:

$$\Gamma; \mathcal{H} \vdash_{(\Sigma, \mathcal{A})} \Phi$$

$$\frac{\Gamma \rhd_\Sigma M_1 : A_1 \quad \ldots \quad \Gamma \rhd_\Sigma M_n : A_n \quad P : \langle A_1, \ldots, A_n \rangle \in \mathcal{P}}{\Gamma \rhd_\Sigma P(M_1, \ldots, M_n) \text{ prop}}$$

$$\frac{\Gamma \rhd_\Sigma \Phi \text{ prop} \quad \Gamma \rhd_\Sigma \Psi \text{ prop}}{\Gamma \rhd_\Sigma \Phi \supset \Psi \text{ prop}} \qquad \frac{\Gamma, x : A \rhd_\Sigma \Phi \text{ prop}}{\Gamma \rhd_\Sigma \forall x : A.\, \Phi \text{ prop}}$$

Figure 5: Well-formedness rules for formulae.

$$\text{Ax} \quad \frac{\Phi \in \mathcal{A}}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi} \qquad \text{Ass} \quad \frac{\Phi \in \mathcal{H}}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi}$$

$$\text{Sub} \quad \frac{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi[M/x] \quad M =_{\beta\eta} N}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi[N/x]}$$

$$\supset \text{I} \quad \frac{\Gamma; \mathcal{H}, \Phi \vdash_{(\Sigma,\mathcal{A})} \Psi}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi \supset \Psi} \qquad \supset \text{E} \quad \frac{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi \supset \Psi \quad \Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Psi}$$

$$\forall\text{I} \quad \frac{\Gamma, x : A; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \forall x : A.\, \Phi} \qquad \forall\text{E} \quad \frac{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \forall x : A.\, \Phi \quad \Gamma \rhd_\Sigma M : A}{\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi[M/x]}$$

Restriction on $\forall$I: x does not occur free in $\mathcal{H}$.

Figure 6: Rules for meta-logical consequence.

relates Γ, $\mathcal{H}$ and Φ where $\mathcal{H}$ is a set of formulae, the *hypotheses*, and each formula in $\mathcal{H} \cup \{\Phi\}$ is well-formed in Γ and Σ. This relation is given by the formal system of Figure 6.

In the sequel we shall require the following elementary derived result about consequence.

Lemma 2.1 (weakening) *If $\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi$ and all formulae in $\mathcal{H}'$ are well-formed in $\Gamma \cup \Gamma'$ then $\Gamma, \Gamma'; \mathcal{H}, \mathcal{H}' \vdash_{(\Sigma,\mathcal{A})} \Phi$.*

The easy proof, by induction on the structure of derivations, is omitted.

3 Semantics

As the meta-logic is intuitionistic with quantification over all higher types, we seek a semantics in terms of Kripke models in which all typed lambda terms can be interpreted at each world. The Kripke lambda models of Mitchell and Moggi [9] are thus a natural choice. Although we follow their paper quite closely, the reader is advised that some of our notation and terminology differs from that of [9].

An (extensional, Kripke, *$\mathcal{T}$-$\mathcal{P}$-)prestructure* is a sextuple:

$$\langle W, \leq, \{[\![A]\!]_w\}, \{[\![P]\!]_w\}, \{\epsilon_w^{AB}\}, \{i_{ww'}^A\}\rangle$$

where:

- W is a set of *worlds* partially orded by $\leq$.
- $\{[\![A]\!]_w\}$ is a family of sets, $[\![A]\!]_w$, indexed by types, A, and worlds, w.
- $\{[\![P]\!]_w\}$ is a family of relations, $[\![P]\!]_w \subseteq [\![A_1]\!]_w \times \ldots \times [\![A_n]\!]_w$, indexed by predicate symbols, P, with declarations, $P:\langle A_1, \ldots, A_n\rangle$, in $\mathcal{P}$ and worlds, w.
- $\{\epsilon_w^{AB}\}$ is a family of functions, $\epsilon_w^{AB} : [\![A \to B]\!]_w \times [\![A]\!]_w \longrightarrow [\![B]\!]_w$, indexed by pairs of types, A, B, and worlds, w.
- $\{i_{ww'}^A\}$ is a family of functions, $i_{ww'}^A : [\![A]\!]_w \to [\![A]\!]_{w'}$, indexed by types, A, and pairs of worlds, $w \leq w'$.

subject to the conditions given below. In these (and henceforth) we adopt the following notational conventions. When $f \in [\![A \to B]\!]_w$ and $a \in [\![A]\!]_w$, we write $f(a)$ for $\epsilon_w^{AB}(f, a)$. When $a_w \in [\![A]\!]_w$ and $w \leq w'$, we write $a_{w'}$ for $i_{ww'}^A(a_w)$.

The conditions are:

identity: For all worlds w, i_{ww}^A is the identity.

composition: For all $w \leq w' \leq w''$, $i_{w'w''}^A \circ i_{ww'}^A = i_{ww''}^A$.

naturality: For all $w \leq w'$, $i_{ww'}^B \circ \epsilon_w^{AB} = \epsilon_{w'}^{AB} \circ (i_{ww'}^{A\to B} \times i_{ww'}^A)$.

extensionality: If $f_w, g_w \in [\![A \to B]\!]_w$ and, for all $w' \geq w$, for all $a \in [\![A]\!]_{w'}$:

$$f_{w'}(a) = g_{w'}(a)$$

then $f_w = g_w$.

persistency: If $[\![P]\!]_w(a_{1w},\ldots,a_{nw})$ then, for all $w' \geq w$, $[\![P]\!]_{w'}(a_{1w'},\ldots,a_{nw'})$.

Thus a prestructure is an "extensional Kripke applicative structure" in the terminology of [9], together with an extra parameter, $\{[\![P]\!]_w\}$, used for interpreting the predicates of the logic.

A *partial element*, p, of type A in a prestructure is given by an upper-closed subset $\mathrm{dom}(p) \subseteq W$, its *domain*, and a family of elements, $\{p_w\}$, indexed by worlds $w \in \mathrm{dom}(p)$ such that for all $w' \geq w \in \mathrm{dom}(p)$, $p_w \in [\![A]\!]_w$ and $i^A_{ww'}(p_w) = p_{w'}$. Given $p_w \in [\![A]\!]_w$, we write p for the induced partial element of type A with domain $\{w' \mid w \leq w'\}$ given by the elements $p_{w'} \in [\![A]\!]_{w'}$. A *global element* is a partial element, p, for which $\mathrm{dom}(p) = W$.

A *structure* is an 8-tuple:

$$\langle W, \leq, \{[\![A]\!]_w\}, \{[\![P]\!]_w\}, \{\epsilon^{A_1A_2}_w\}, \{i^A_{ww'}\}, \{K^{AB}_w\}, \{S^{ABC}_w\}\rangle$$

which is a prestructure extended by $\{K^{AB}_w\}$ and $\{S^{ABC}_w\}$ where:

- K^{AB} is a global element of type $A \to B \to A$ such that, for all worlds w, for all $a \in [\![A]\!]_w$, for all $b \in [\![B]\!]_w$, $K^{AB}_w(a)(b) = a$.
- S^{ABC} is a global element of type $(A \to B \to C) \to (A \to B) \to A \to C$ such that, for all worlds w, for all $f \in [\![A \to B \to C]\!]_w$, for all $g \in [\![A \to B]\!]_w$, for all $a \in [\![A]\!]_w$, $S^{ABC}_w(f)(g)(a) = f(a)(g(a))$.

A Σ-*structure* is a 9-tuple:

$$\langle W, \leq, \{[\![A]\!]_w\}, \{[\![P]\!]_w\}, \{[\![c]\!]_w\}, \{\epsilon^{A_1A_2}_w\}, \{i^A_{ww'}\}, \{K^{AB}_w\}, \{S^{ABC}_w\}\rangle$$

which is a structure extended by $\{[\![c]\!]_w\}$ where:

- $\{[\![c]\!]_w\}$ is a family of global elements, $[\![c]\!]$, of type A indexed by constants, c, with declarations, $c\!:\!A$, in Σ.

Henceforth we refer to a Σ-structure as $(W, \leq)$ leaving the other components implicit.

An *environment*, ρ, is a function from variables to partial elements. We say that ρ *interprets* Γ *at* w if, for all $x\!:\!A \in \Gamma$, $\rho(x)$ is a partial element of type A with $w \in \mathrm{dom}(\rho(x))$. Clearly if ρ interprets Γ at w and $w' \geq w$ then ρ interprets Γ at w' too. Also any environment interprets the empty context at any world. Given an environment ρ and an element $a_w \in [\![A]\!]_w$ we write $\rho[x := a]$ for the environment that agrees with ρ on variables other than x and which assigns the induced partial element a to x. If ρ interprets Γ at w and $a_w \in [\![A]\!]_w$ then clearly $\rho[x := a]$ interprets $\Gamma, x\!:\!A$ at w.

If M has type A in Γ, and ρ interprets Γ at w, then the interpretation, $[\![M]\!]^\rho_w \in [\![A]\!]_w$, of M by ρ at w is defined inductively on the structure of M by:

$$\begin{array}{rcl}
[\![c]\!]^\rho_w & = & [\![c]\!]_w \\
[\![x]\!]^\rho_w & = & \rho(x)_w \\
[\![\lambda x\!:\!A.\,M]\!]^\rho_w & = & \text{the unique } f_w \in [\![A \to B]\!]_w \text{ (where } A \to B \text{ is the type of} \\
 & & \lambda x\!:\!A.\,M \text{ in } \Gamma\text{) such that, for all } w' \geq w, \text{ for all } a_{w'} \in [\![A]\!]_{w'}, \\
 & & f_{w'}(a_{w'}) = [\![M]\!]^{\rho[x:=a]}_{w'} \\
[\![M(N)]\!]^\rho_w & = & [\![M]\!]^\rho_w([\![N]\!]^\rho_w)
\end{array}$$

As in [10], the existence of the f_w required in the $\lambda x : A.\,M$ clause is given by the S and K combinators, and its uniqueness is guaranteed by extensionality. Clearly if M is well-typed in the empty context then the value of $[\![M]\!]^\rho_w$ is independent of ρ, so we just write $[\![M]\!]_w$.

We now give some lemmas concerning the interpretation of terms in Σ-structures.

Lemma 3.1 *If $\Gamma \rhd_\Sigma M{:}A$, ρ interprets Γ at w and $w \leq w'$, then $i^A_{ww'}(\llbracket M \rrbracket^\rho_w) = \llbracket M \rrbracket^\rho_{w'}$.*

Lemma 3.2 *If $\Gamma, x : A \rhd_\Sigma M : B$, $\Gamma \rhd_\Sigma N : A$ and ρ interprets Γ at w, then $\llbracket M[N/x] \rrbracket^\rho_w = \llbracket M \rrbracket^{\rho[x:=\llbracket N \rrbracket^\rho]}_w$.*

Lemma 3.3 *If $\Gamma \rhd_\Sigma M{:}A$, $M =_{\beta\eta} N$, and ρ interprets Γ at w, then $\llbracket M \rrbracket^\rho_w = \llbracket N \rrbracket^\rho_w$.*

The first two lemmas are proved by straightforward inductions on the structure of M. The third is proved by an induction on derivations (in the usual formal system for $\beta\eta$-equality) of $M =_{\beta\eta} N$, using Lemma 3.2 in the verification β-equality.

If Φ is well-formed in Γ, and ρ interprets Γ at w, then the "forcing" relation $w \models_\rho \Phi$ is defined inductively on the structure of Φ by:

$$
\begin{array}{lll}
w \models_\rho P(M_1, \ldots, M_n) & \text{iff} & \llbracket P \rrbracket_w(\llbracket M_1 \rrbracket^\rho_w, \ldots, \llbracket M_n \rrbracket^\rho_w) \\
w \models_\rho \Phi \supset \Psi & \text{iff} & \text{for all } w' \geq w, \text{ if } w' \models_\rho \Phi \text{ then } w' \models_\rho \Psi \\
w \models_\rho \forall x{:}A.\, \Phi & \text{iff} & \text{for all } w' \geq w, \text{ for all } a_{w'} \in \llbracket A \rrbracket_{w'},\ w' \models_{\rho[x:=a]} \Phi
\end{array}
$$

If $\mathcal{H}$ is a set of formulae, each well-formed in Γ, and ρ interprets Γ at w then we write $w \models_\rho \mathcal{H}$ to mean that $w \models_\rho \Phi$, for all $\Phi \in \mathcal{H}$. If Φ is well-formed in the empty context then whether $w \models_\rho \Phi$ holds or not is independent of ρ, so we write $w \models \Phi$. We write $(W, \leq) \models \Phi$ to mean, for all $w \in W$, $w \models \Phi$.

The lemmas below give basic properties of the forcing relation.

Lemma 3.4 *If $w \models_\rho \Phi$ and $w \leq w'$ then $w' \models_\rho \Phi$.*

Lemma 3.5 *If $\Gamma \rhd_\Sigma M : A$ and ρ interprets Γ at w, then $w \models_\rho \Phi[M/x]$ if and only if $w \models_{\rho[x:=\llbracket M \rrbracket^\rho]} \Phi$.*

Both these lemmas are proved by induction on the structure of Φ. The base case of the first uses Lemma 3.1 together with the persistency property of the structure.

A $(\Sigma, \mathcal{A})$-*model* is a Σ-structure, $(W, \leq)$, such that, for all $\Phi \in \mathcal{A}$, $(W, \leq) \models \Phi$.

Theorem 3.6 (soundness and completeness) *The two statements below are equivalent.*

1. *$\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi$.*

2. *For all $(\Sigma, \mathcal{A})$-models $(W, \leq)$, for all $w \in W$, for all ρ interpreting Γ at w, if $w \models_\rho \mathcal{H}$ then $w \models_\rho \Phi$.*

We devote the rest of this section to an outline of the proof of the theorem.

Soundness (ie. *1.* $\Longrightarrow$ *2.*) is proved by a straightforward induction on the derivation of $\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi$. Lemma 3.5 is used in the verification of both the $\forall$E rule and (in conjunction with Lemma 3.3) the Sub rule.

To prove completeness we construct a particular model, $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$ based on the presentation $(\Sigma, \mathcal{A})$. The set of worlds is just:

$$W_{(\Sigma,\mathcal{A})} \quad = \quad \{(\Gamma, \mathcal{H}) \mid \text{every formula in } \mathcal{H} \text{ is well-formed in } \Gamma\}$$

Its partial order is given by:

$$(\Gamma, \mathcal{H}) \leq_{(\Sigma,\mathcal{A})} (\Gamma', \mathcal{H}') \quad \text{iff} \quad \Gamma \subseteq \Gamma' \text{ and } \mathcal{H} \subseteq \mathcal{H}'$$

The interpretations of the other components of the Σ-structure are:

$$\begin{array}{rcl} [\![A]\!]_{(\Gamma,\mathcal{H})} & = & \{M \mid \Gamma \rhd_\Sigma M\!:\!A,\ M \text{ is in long-}\beta\eta \text{ normal form}\} \\ [\![P]\!]_{(\Gamma,\mathcal{H})}(M_1,\ldots,M_n) & \text{iff} & \Gamma;\mathcal{H} \vdash_{(\Sigma,\mathcal{A})} P(M_1,\ldots,M_n) \\ [\![c]\!]_{(\Gamma,\mathcal{H})} & = & \beta\eta(c) \\ \epsilon^{AB}_{(\Gamma,\mathcal{H})}(M,N) & = & \beta\eta(M(N)) \\ i^{A}_{(\Gamma,\mathcal{H})(\Gamma',\mathcal{H}')}(M) & = & M \\ K^{AB}_{(\Gamma,\mathcal{H})} & = & \beta\eta(\lambda x\!:\!A.\,\lambda y\!:\!B.\,x) \\ S^{ABC}_{(\Gamma,\mathcal{H})} & = & \beta\eta(\lambda x\!:\!A\to B\to C.\,\lambda y\!:\!A\to B.\,\lambda z\!:\!A.\,x(z)(y(z))) \end{array}$$

Proposition 3.7 *$(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$ is a Σ-structure.*

Proof. First, all the components are clearly well-defined, in particular $i^{A}_{(\Gamma,\mathcal{H})(\Gamma',\mathcal{H}')}$ is, because long-$\beta\eta$ normal forms are preserved by context extensions. So we need only check the various conditions. Identity, composition and naturality all follow trivially from the definitions of $i^{A}_{(\Gamma,\mathcal{H})(\Gamma',\mathcal{H}')}$ and $\epsilon^{AB}_{(\Gamma,\mathcal{H})}$. Persistency is an immediate consequence of weakening (Lemma 2.1). To prove extensionality: suppose $M, M' \in [\![A \to B]\!]_{(\Gamma,\mathcal{H})}$ are such that, for all $\Gamma' \supseteq \Gamma$, for all $\mathcal{H}' \supseteq \mathcal{H}$, for all $N \in [\![A]\!]_{(\Gamma',\mathcal{H}')}$, $\beta\eta(M(N)) = \beta\eta(M'(N))$. Then in particular, for $x \in [\![A]\!]_{(\Gamma\cup\{x:A\},\mathcal{H})}$ (where x does not appear in Γ), $\beta\eta(M(x)) = \beta\eta(M'(x))$, and so $M(x) =_{\beta\eta} M'(x)$. But then $\lambda x\!:\!A.\,M(x) =_{\beta\eta} \lambda x\!:\!A.\,M'(x)$ and so (by η-equality) $M =_{\beta\eta} M'$. But M and M' are both in long-$\beta\eta$ normal form, so $M = M'$, proving extensionality. It remains to check that K^{AB} and S^{ABC} satisfy the required equalities, but these follow easily from the β-rule of typed lambda-calculus. □

Given a context Γ we construct a canonical environment $\rho(\Gamma)$ as follows. If, for some A, $x : A \in \Gamma$ then $\rho(\Gamma)(x)$ is the partial element with domain $\{(\Gamma',\mathcal{H}) \mid \Gamma \subseteq \Gamma'\}$ given by $\rho(\Gamma)(x)_{(\Gamma',\mathcal{H})} = \beta\eta(x)$. Otherwise, if there is no A such that $x\!:\!A \in \Gamma$ then $\rho(\Gamma)(x)$ is the trivial partial element with domain $\emptyset$. Clearly, for all $\Gamma' \supseteq \Gamma$, $\rho(\Gamma)$ interprets Γ at $(\Gamma',\mathcal{H})$.

We can now state the basic properties of $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$.

Lemma 3.8 *If $\Gamma \rhd_\Sigma M\!:\!A$ and $\Gamma \subseteq \Gamma'$ then $[\![M]\!]^{\rho(\Gamma)}_{(\Gamma',\mathcal{H})} = \beta\eta(M)$.*

Proof. By an easy induction on the structure of M. □

Lemma 3.9 *$\Gamma;\mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi$ if and only if $(\Gamma,\mathcal{H}) \models_{\rho(\Gamma)} \Phi$.*

Proof. By induction on the structure of Φ. The cases are:

$P(M_1,\ldots,M_n)$. Immediate from definition of $[\![P]\!]_{(\Gamma,\mathcal{H})}$.

$\Phi \supset \Psi$. Suppose $\Gamma;\mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi \supset \Psi$. Take any $\Gamma' \supseteq \Gamma$ and $\mathcal{H}' \supseteq \mathcal{H}$ such that $(\Gamma',\mathcal{H}') \models_{\rho(\Gamma)} \Phi$. Clearly $(\Gamma',\mathcal{H}') \models_{\rho(\Gamma')} \Phi$. So, by the induction hypothesis, $\Gamma';\mathcal{H}' \vdash_{(\Sigma,\mathcal{A})} \Phi$. Now $\Gamma';\mathcal{H}' \vdash_{(\Sigma,\mathcal{A})} \Phi \supset \Psi$, so $\Gamma';\mathcal{H}' \vdash_{(\Sigma,\mathcal{A})} \Psi$. Then, by the induction hypothesis again, $(\Gamma',\mathcal{H}') \models_{\rho(\Gamma')} \Psi$, so clearly $(\Gamma',\mathcal{H}') \models_{\rho(\Gamma)} \Psi$ (all the variables in Ψ are in Γ). Thus we have shown that $(\Gamma,\mathcal{H}) \models_{\rho(\Gamma)} \Phi \supset \Psi$ as required.

Conversely, suppose $(\Gamma,\mathcal{H}) \models_{\rho(\Gamma)} \Phi \supset \Psi$. Now $\Gamma;\mathcal{H},\Phi \vdash_{(\Sigma,\mathcal{A})} \Phi$, so, by the induction hypothesis, $(\Gamma,\mathcal{H}\cup\{\Phi\}) \models_{\rho(\Gamma)} \Phi$. But then, by the supposition, $(\Gamma,\mathcal{H}\cup\{\Phi\}) \models_{\rho(\Gamma)} \Psi$. So, again by the induction hypothesis, $\Gamma;\mathcal{H},\Phi \vdash_{(\Sigma,\mathcal{A})} \Psi$. Thus clearly $\Gamma;\mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi \supset \Psi$.

$\forall x : A.\ \Phi$. Suppose $\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \forall x : A.\ \Phi$. Take any $\Gamma' \supseteq \Gamma$, $\mathcal{H}' \supseteq \mathcal{H}$ and $M \in [\![A]\!]_{(\Gamma',\mathcal{H}')}$. Thus $\Gamma' \rhd_\Sigma M : A$ and $\Gamma'; \mathcal{H}' \vdash_{(\Sigma,\mathcal{A})} \forall x : A.\ \Phi$, so $\Gamma'; \mathcal{H}' \vdash_{(\Sigma,\mathcal{A})} \Phi[M/x]$. Now, by the induction hypothesis, $(\Gamma', \mathcal{H}') \models_{\rho(\Gamma')} \Phi[M/x]$, so clearly $(\Gamma', \mathcal{H}') \models_{\rho(\Gamma)} \Phi[M/x]$. Thus, by Lemma 3.5, $(\Gamma', \mathcal{H}') \models_{\rho(\Gamma)[x := [\![M]\!]^{\rho(\Gamma)}]} \Phi$. Then, by Lemma 3.8, $(\Gamma', \mathcal{H}') \models_{\rho(\Gamma)[x:=M]} \Phi$ (as M is necessarily in long-$\beta\eta$ normal form). We have therefore shown that $(\Gamma, \mathcal{H}) \models_{\rho(\Gamma)} \forall x : A.\ \Phi$ as required.

Conversely, suppose $(\Gamma, \mathcal{H}) \models_{\rho(\Gamma)} \forall x : A.\ \Phi$. Then $\beta\eta(x) \in [\![A]\!]_{(\Gamma \cup \{x:A\},\mathcal{H})}$. So $(\Gamma \cup \{x : A\}, \mathcal{H}) \models_{\rho(\Gamma \cup \{x:A\})} \Phi$. Then, by the induction hypothesis, $\Gamma, x : A; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi$. So clearly $\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \forall x : A.\ \Phi$.

□

Corollary 3.10 $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$ *is a* $(\Sigma, \mathcal{A})$*-model.*

Proof. Immediate. □

We now finish the proof of completeness. Suppose that *2.* holds. Then in particular $(\Gamma, \mathcal{H}) \models_{\rho(\Gamma)} \Phi$. So, by Lemma 3.9, $\Gamma; \mathcal{H} \vdash_{(\Sigma,\mathcal{A})} \Phi$ as required.

It is worth comparing our use of Kripke lambda models with the use in [9]. In both cases the role of the partial order is to model intuitionistic entailment. But there are differences in emphasis due to the different logical languages considered. In [9], although it is remarked that the models interpret full intuitionistic predicate logic with quantification over all higher types, only the interpretation of equations is given explicitly. This is because their interest is in obtaining a completeness theorem for the usual equational consequence relation of the typed lambda-calculus when empty types are permitted. In this paper we too are not making full use of the scope of the models. We consider only a fragment of the full intuitionistic logic, and we have no equality predicate in the logic. The absence of equality means that the definition of model could be simplified in various ways. For example, it would be possible to insist that the coercions, $i^A_{ww'}$, are injections. However, such restrictions are unnatural (we shall have use for a model in which the coercions are not inclusions). Furthermore, we wish to keep the definition of model in its full generality to allow the logic to be extended with equality (and indeed the other connectives) if desired. Note that a completeness proof for the logic with equality would require a model built out of equivalence classes of terms (as in [9]), for unique long-$\beta\eta$ normal forms would no longer be available.

4 Logics as consequence relations

The emphasis of this paper is to be on the use of the meta-logic as a logical framework. In order to investigate this formally, we need to consider what it means to encode a logic. So, as a preliminary step, it is necessary that we establish a precise notion of logic.

We consider a logic to be a pair, $(\mathcal{L}, \vdash)$, where $\mathcal{L}$ is a countable set (of *formulae*) and $\vdash \subseteq \wp(\mathcal{L}) \times \mathcal{L}$ is a (possibly non-compact) *consequence relation*. This viewpoint is over simplistic on (at least) two counts. First, there is no analysis of the underlying syntax of formulae. Second, many well-known logics have consequence relations that are not ordinary. One might also criticise on the grounds that proof-theory and semantics ought to be important components in the definition of a logic. But generally the consequence relation really is the fundamental characteristic of a logic. Though one is often interested in many different proof systems and semantics for the same logic, the consequence relation remains invariant. However, it is to be hoped that natural extensions of the above notion of logic, taking syntax, proof-theory and semantics into account, will arise.

Actually, even according to the above approach, semantic issues are not entirely overlooked. It turns out that the consequence relations we consider correspond exactly to the entailment relations of languages with a certain kind of "validity-style" semantics. It is this correspondence that motivates the choice of definition of consequence relation. It is fair to say that the semantic correspondence is not very deep, but this is inevitable when we work with such a superficial notion of syntax.

For the moment we suppose a given $\mathcal{L}$ and an *arbitrary* $\vdash \subseteq \wp(\mathcal{L}) \times \mathcal{L}$. We use φ, ψ and θ to range over elements of $\mathcal{L}$, and Δ to range over subsets of $\mathcal{L}$. Every set of formulae, Δ, determines a *theory*, $Th_{\vdash}(\Delta)$, defined by:

$$Th_{\vdash}(\Delta) \quad = \quad \{\varphi \mid \Delta \vdash \varphi\}$$

We define $Theories_{\vdash} = \{Th_{\vdash}(\Delta) \mid \Delta \subseteq \mathcal{L}\}$, the set of theories of $\vdash$. Note that $\mathcal{L}$ itself may well be a theory (it *will* be whenever $\vdash$ is a consequence relation). It will be a recurring theme that the "inconsistent" theory will not be distinguished, thus enabling many definitions to be more uniform. If desired, all definitions could be easily repaired to treat the inconsistent theory separately.

Definition 4.1 (consequence relation) *$\vdash$ is a consequence relation if it satisfies the following two conditions.*

CR1 *If $\varphi \in \Delta$ then $\Delta \vdash \varphi$.*

CR2 *If $\Delta' \vdash \varphi$ and, for all $\varphi' \in \Delta'$, $\Delta \vdash \varphi'$ then $\Delta \vdash \varphi$.*

A consequence relation, $\vdash$, is said to be *compact* if, whenever $\Delta \vdash \varphi$, there is a finite $\Delta' \subseteq \Delta$ such that $\Delta' \vdash \varphi$.

Definition 4.2 (validity-style semantics) *A validity-style semantics for $\vdash$ is a pair, $(Mod, \models)$, where Mod is a set (of "models") and $\models \subseteq Mod \times \mathcal{L}$ is a ("satisfaction") relation satisfying:*

$$\Delta \vdash \varphi \quad \textit{iff} \quad \textit{for all } \mathcal{M} \in Mod, \textit{ if, for all } \varphi' \in \Delta,\ \mathcal{M} \models \varphi' \textit{ then } \mathcal{M} \models \varphi$$

Theorem 4.3 *The following are equivalent.*

1. *$\vdash$ is a consequence relation.*
2. *$Th_{\vdash}$ is a closure operation on $\wp(\mathcal{L})$.*
3. *$\vdash$ has a validity-style semantics.*

Proof. It is easy to check that *1.* is equivalent to *2.* and that *3.* implies *1.*. To see that *1.* implies *3.*, assume that $\vdash$ is a consequence relation. We now exhibit a validity-style semantics for $\vdash$. The set of models is:

$$Mod \quad = \quad \wp(\mathcal{L})$$

and the satisfaction relation is defined by:

$$\Delta \models \varphi \quad \text{iff} \quad \Delta \vdash \varphi$$

It is easy to check that this is indeed a validity-style semantics for $\vdash$. $\Box$

Theorem 4.3 gives the promised correspondence between consequence relations and semantics. One way in which it is rather shallow is that it says nothing about the existence or nature of "interesting" models.

Henceforth we take $\vdash$ to be a consequence relation over $\mathcal{L}$, and $(Mod, \models)$ to be a chosen validity-style semantics for $\vdash$.

We now fix some notation that we shall use later on. Each set of formulae, Δ, determines a set of models, $Mods_{\models}(\Delta)$, defined by:

$$Mods_{\models}(\Delta) \quad = \quad \{\mathcal{M} \mid \text{for all } \varphi \in \Delta,\ \mathcal{M} \models \varphi\}$$

We say that $\mathcal{M} \in Mods_{\models}(\Delta)$ is *a model of* Δ. In fact the set of models is determined by the theory of Δ, that is, $Mods_{\models}(\Delta) = Mods_{\models}(Th_{\vdash}(\Delta))$. Note again that the inconsistent theory, $\mathcal{L}$, is not treated specially in that $Mods_{\models}(\mathcal{L})$ may be non-empty. Conversely, each set of models, $\mathcal{S}$, determines a theory, $Th_{\models}(\mathcal{S})$, defined by:

$$Th_{\models}(\mathcal{S}) \quad = \quad \{\varphi \mid \text{for all } \mathcal{M} \in \mathcal{S},\ \mathcal{M} \models \varphi\}$$

It is easy to check that $Th_{\models}(\mathcal{S})$ is indeed a theory.

The definition of consequence relation above is standard except that we do not follow the common practice of assuming compactness. In essence it is equivalent to the relations considered ever since Tarski introduced his consequence operators [15]. There are two reasons for not assuming compactness. One is that many interesting non-compact logics exist. The second is that without compactness we obtain the equivalences of Theorem 4.3. The notion of validity-style semantics used in the theorem is a familiar one from abstract model theory [2]. The correspondence between consequence relations and semantics is similar to Scott's truth-valuation motivation for his two-sided consequence relations [14].

5 Encoding logics in the meta-logic

We now consider how the meta-logic can be used to encode the consequence relation of a logic. An *encoding* of $(\mathcal{L}, \vdash)$ is given by a presentation $(\mathcal{T}, \mathcal{P}, \Sigma, \mathcal{A})$ containing a distinguished type constant, $o \in \mathcal{T}$, and a distinguished predicate, *true*, with declaration $true : \langle o \rangle \in \mathcal{P}$, together with a bijective function:

$$(\cdot)^* : \mathcal{L} \to \{M \mid \rhd_{\Sigma} M : o,\ M \text{ is in long-}\beta\eta \text{ normal form}\}$$

mapping formulae of the object-logic to their representing terms in the meta-logic.

For an encoding to be of any use, it must respect the consequence relation of the encoded logic. The encoding is said to be *full* if:

$$\Delta \vdash \varphi \quad \text{implies} \quad true(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} true(\varphi^*)$$

where $true(\Delta^*)$ denotes the set $\{true(\psi^*) \mid \psi \in \Delta\}$. It is said to be *faithful* if:

$$true(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} true(\varphi^*) \quad \text{implies} \quad \Delta \vdash \varphi$$

An encoding which is both full and faithful is said to be *adequate*. Adequacy is a minimal correctness condition. Without it there is a mismatch between the consequence relation of the logic and its representation in the meta-logic.

Clearly any logic that has an adequate encoding must be compact (this is because of the evident compactness of meta-logical consequence). In fact the converse holds: any compact logic has an adequate encoding. We do not go into details as the encoding is obtained in a wholly unilluminating way (for example one includes separate term constants for each $\varphi \in \mathcal{L}$) and is in general infinite. A more interesting result might be obtainable by considering some suitable notion of effective consequence relation. The compactness of such a consequence relation ought to follow from an adaptation of the Myhill-Sheperdson theorem of recursive function theory. The desired theorem would be that the notion of effective consequence relation coincides with that of consequence relation with an adequate *finite* presentation.

The two halves of adequacy each correspond to conditions on the class of $(\Sigma, \mathcal{A})$-models. Given a $(\Sigma, \mathcal{A})$-model, $(W, \leq)$, define $\mathit{form} : W \to \wp(\mathcal{L})$ by:

$$\mathit{form}(w) \quad = \quad \{\varphi \mid w \models \mathit{true}(\varphi^*)\}$$

We write $\mathit{form}(W)$ for the set $\{\mathit{form}(w) \mid w \in W\}$.

Proposition 5.1 *The following are equivalent:*

1. *The encoding is full.*

2. *For all $(\Sigma, \mathcal{A})$-models $(W, \leq)$, $\mathit{form}(W) \subseteq \mathit{Theories}_\vdash$.*

Proof. Suppose the encoding is full. Let $(W, \leq)$ be any $(\Sigma, \mathcal{A})$-model, and take any $w \in W$. We must show that $\mathit{form}(w) \in \mathit{Theories}_\vdash$. Suppose then that $\mathit{form}(w) \vdash \varphi$. By fullness, $\mathit{true}(\mathit{form}(w)^*) \vdash_{(\Sigma,\mathcal{A})} \mathit{true}(\varphi^*)$. But, by definition of $\mathit{form}(w)$, for all $\psi \in \mathit{form}(w)$, $w \models \mathit{true}(\psi^*)$. So, by the soundness of the meta-logic, $w \models \mathit{true}(\varphi^*)$. But then $\varphi \in \mathit{form}(w)$. So indeed $\mathit{form}(w) \in \mathit{Theories}_\vdash$.

Conversely, suppose that, for all $(\Sigma, \mathcal{A})$-models $(W, \leq)$, $\mathit{form}(W) \subseteq \mathit{Theories}_\vdash$. Suppose further that $\Delta \vdash \varphi$. We must show that $\mathit{true}(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} \mathit{true}(\varphi^*)$. For this we use the model, $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$, constructed in the proof of completeness in section 3. By Corollary 3.10, we know this is indeed a $(\Sigma, \mathcal{A})$-model. Consider the world $(\emptyset, \mathit{true}(\Delta^*))$. By the initial supposition, $\mathit{form}((\emptyset, \mathit{true}(\Delta^*))) \in \mathit{Theories}_\vdash$. But clearly $\Delta^* \subseteq \mathit{form}((\emptyset, \mathit{true}(\Delta^*)))$, so $\varphi \in \mathit{form}((\emptyset, \mathit{true}(\Delta^*)))$ (as any theory is closed under consequence). Therefore $(\emptyset, \mathit{true}(\Delta^*)) \models \mathit{true}(\varphi^*)$. So, by the definition of $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$, $\mathit{true}(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} \mathit{true}(\varphi^*)$. $\square$

Proposition 5.2 *The following are equivalent:*

1. *The encoding is faithful.*

2. *There exists a $(\Sigma, \mathcal{A})$-model, $(W, \leq)$, such that $\mathit{Theories}_\vdash \subseteq \mathit{form}(W)$.*

Proof. Suppose the encoding is faithful. We will show that the $(\Sigma, \mathcal{A})$-model $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$ has the required property. Take any $T \in \mathit{Theories}_\vdash$. We must show that there exists $(\Gamma, \mathcal{H}) \in W_{(\Sigma,\mathcal{A})}$ with $\mathit{form}((\Gamma, \mathcal{H})) = T$. For this, we take the world, $(\emptyset, \mathit{true}(T^*))$. Clearly (from the definition of $(W_{(\Sigma,\mathcal{A})}, \leq_{(\Sigma,\mathcal{A})})$) $T \subseteq \mathit{form}((\emptyset, \mathit{true}(T^*)))$. We now show that $\mathit{form}((\emptyset, \mathit{true}(T^*))) \subseteq T$. Suppose that $\varphi \in \mathit{form}((\emptyset, \mathit{true}(T^*)))$. Then $(\emptyset, \mathit{true}(T^*)) \models \mathit{true}(\varphi^*)$, so $\mathit{true}(T^*) \vdash_{(\Sigma,\mathcal{A})} \mathit{true}(\varphi^*)$. Therefore, by faithfulness, $T \vdash \varphi$. Thus indeed $\varphi \in T$ (as T is closed under consequence).

Conversely, suppose there exists a $(\Sigma, \mathcal{A})$-model, $(W, \leq)$, such that $\mathit{Theories}_\vdash \subseteq \mathit{form}(W)$. Suppose further that $\mathit{true}(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} \mathit{true}(\varphi^*)$. We must show that $\Delta \vdash \Phi$. Let $w \in W$ be such that $form(w) = Th_{\vdash}(\Delta)$ (such a w is guaranteed to exist by the assumed property of $(W, \leq)$). Clearly, for all $\psi \in \Delta$, $w \models true(\psi^*)$. So, by soundness, $w \models true(\varphi^*)$. But then $\varphi \in form(w)$ so $\varphi \in Th_{\vdash}(\Delta)$. Thus $\Delta \vdash \varphi$ as required. □

The above propositions give the basic connections between the semantics of the meta-logic and the use of the meta-logic as a logical framework. Due to the universal quantification over models, Proposition 5.1 is of mainly formal interest. In contrast, Proposition 5.2 is genuinely useful and will provide an important tool in the sequel.

One important use of Proposition 5.2 will be to establish the adequacy of encodings. Proving adequacy is a three stage process. First, it is necessary to define the encoding function and show that it is indeed a bijection as required. This task can only be done syntactically, usually by an easy induction on the long-$\beta\eta$ normal forms of appropriate types. Second, it is necessary to establish fullness. Normally the easiest way to do this is to start with a proof system for the encoded logic and map proofs of logical consequence in this system to proofs of the representation of the consequence in the meta-logic. Again this step is usually straightforward. Lastly, faithfulness remains to be proved. The standard technique is to show that the mapping of proofs used in establishing fullness is itself a bijection onto meta-logical proofs in a certain normal form. However, in order to do this it is necessary that the meta-theory does stand in the right relationship to the proof system of the object-logic. Often this will not be the case, either because of the inability of the meta-logic to exactly mimic the proof system or, more positively, because the meta-theory allows other (still valid) inferences not directly avilable in the original proof system (see the next section). No doubt, even in such difficult cases, it is still possible to prove faithfulness by a syntactic argument. However, Proposition 5.2 offers an alternative approach. It is enough to construct a $(\Sigma, \mathcal{A})$-model with the requisite property (which is usually apparent). These two approaches to proving faithfulness are analogous to the two standard approaches to proving the underivability of a formula in logic using, on the one hand, a normal-form result for derivations and, on the other, a model refuting the formula.

We conclude this section with an example semantic proof of adequacy. We shall show that the presentation, $(\mathcal{T}_A, \mathcal{P}_A, \Sigma_A, \mathcal{A}_Q)$ (the "A" subscript stands for arithmetic, the "Q" subscript stands for system Q), in Figure 7 gives an adequate encoding of the consequence relation, $(\mathcal{L}_A, \vdash_Q)$, of Robinson's arithmetic, Q. The language of Q is that of first-order arithmetic, with terms, t, and formulae, φ, in accordance with the grammar below.

$$\begin{aligned} t &::= x \mid 0 \mid s(t) \mid t_1 + t_2 \mid t_1 \cdot t_2 \\ \varphi &::= t_1 = t_2 \mid \neg\varphi \mid \varphi \Rightarrow \psi \mid \forall x.\, \varphi \end{aligned}$$

For simplicity we just consider two connectives (implication and negation) and one (the universal) quantifier (which is enough as the logic is classical). Again, we assume that the notions of free and bound variable are understood. As usual, a *sentence* is a formula with no free variables. Although we have been informally referring to $\mathcal{L}$ as the formulae of the object-logic, here we take $\mathcal{L}_A$ to be the sentences of arithmetic. $\vdash_Q$ is just classical first-order entailment over $\mathcal{L}_A$ subject to the validity of the axioms for Q. Note that we are considering Q as a logic, and hence as a consequence relation. This is a slight abuse of terminology, Q should really refer to a particular theory: the sentences φ for which $\vdash_Q \varphi$. However, we are more interested in the logic than the theory, as $(\mathcal{T}_A, \mathcal{P}_A, \Sigma_A, \mathcal{A}_Q)$ will turn out to be adequate for logical consequence. It is also of practical interest to encode the consequence relation. For example, we might want to consider derivations from true (or even false) unprovable formulae such as Gödel's sentence or various forms of induction.

$$
\begin{array}{rcl}
\mathcal{T}_A & = & \{\iota, o\} \\
\mathcal{P}_A & = & \{\mathit{true}\!:\!\langle o\rangle\} \\
\Sigma_A & = & \{\Rightarrow\!: o \to o \to o,\ \neg : o \to o,\ \forall : (\iota \to o) \to o,\ = : \iota \to \iota \to o, \\
& & \quad 0 : \iota,\ s : \iota \to \iota,\ + : \iota \to \iota \to \iota,\ \cdot : \iota \to \iota \to \iota\} \\
\mathcal{A}_Q & = & \{\forall \varphi\!:\!o.\ \forall \psi\!:\!o.\ \mathit{true}(\varphi \Rightarrow \psi \Rightarrow \varphi), \\
& & \quad \forall \varphi\!:\!o.\ \forall \psi\!:\!o.\ \forall \theta\!:\!o.\ \mathit{true}((\varphi \Rightarrow \psi \Rightarrow \theta) \Rightarrow (\varphi \Rightarrow \psi) \Rightarrow \varphi \Rightarrow \theta), \\
& & \quad \forall \varphi\!:\!o.\ \forall \psi\!:\!o.\ \mathit{true}((\neg\varphi \Rightarrow \neg\psi) \Rightarrow \psi \Rightarrow \varphi), \\
& & \quad \forall \varphi\!:\!\iota \to o.\ \forall t\!:\!\iota.\ \mathit{true}((\forall x.\ \varphi(x)) \Rightarrow \varphi(t)), \\
& & \quad \forall t\!:\!\iota.\ \mathit{true}(t = t), \\
& & \quad \forall \varphi\!:\!o.\ \forall \psi\!:\!o.\ \mathit{true}(\varphi \Rightarrow \psi) \supset \mathit{true}(\varphi) \supset \mathit{true}(\psi), \\
& & \quad \forall \varphi\!:\!o.\ \forall \psi\!:\!\iota \to o.\ (\forall x\!:\!\iota.\ \mathit{true}(\varphi \Rightarrow \psi(x))) \supset \mathit{true}(\varphi \Rightarrow \forall x.\ \psi(x)), \\
& & \quad \forall \varphi\!:\!\iota \to o.\ \forall t_1\!:\!\iota.\ \forall t_2\!:\!\iota.\ \mathit{true}(t_1 = t_2) \supset \mathit{true}(\varphi(t_1)) \supset \mathit{true}(\varphi(t_2)), \\
& & \quad \mathit{true}(\forall x.\ \forall y.\ s(x) = s(y) \Rightarrow x = y), \\
& & \quad \mathit{true}(\forall x.\ \neg\ s(x) = 0), \\
& & \quad \mathit{true}(\forall x.\ \neg\ x = 0 \Rightarrow \neg\ \forall y.\ \neg\ x = s(y)), \\
& & \quad \mathit{true}(\forall x.\ x + 0 = x), \\
& & \quad \mathit{true}(\forall x.\ \forall y. x + s(y) = s(x + y)), \\
& & \quad \mathit{true}(\forall x.\ x \cdot 0 = 0), \\
& & \quad \mathit{true}(\forall x.\ \forall y.\ x \cdot s(y) = (x \cdot y) + x)\}
\end{array}
$$

Figure 7: A presentation of Robinson's Q.

Before proving adequacy, we give a brief informal explanation of the presentation in Figure 7. There are two type constants: ι for terms and o for formulae. The term constructors in Σ generate the first-order language for arithmetic. Implication is deliberately distinguished notationally from meta-logical implication, and universal quantification is distinguished from its meta-logic counterpart by a different convention of usage. For readability we write $\Rightarrow$, $=$, $+$ and $\cdot$ as infix operators, using standard conventions (for example, $\Rightarrow$ associates to the right). We also write $\forall x.\varphi$ for $\forall(\lambda x\!:\!\iota.\varphi)$. The axioms of the presentation divide naturally into groups. The first five give axioms for classical first-order logic with equality, and the next three give rules of inference (each of these groups subdivides further into a propositional part, a quantificational part and an equality part). The system presented is a quite standard Hilbert system, though our slavery to Okham's razor means there is unlikely to be an identical system in the literature. The next seven axioms are just the axioms of Q. These consist of Peano's six axioms together with the additional axiom, $\forall x.\ \neg\ x = 0 \Rightarrow \neg\ \forall y.\ \neg\ x = s(y)$, necessary in the absence of induction.

Given our notational conventions, it is now a simple matter to define the mapping $(\cdot)^*$ from $\mathcal{L}_A$ to long-$\beta\eta$ normal forms of type o. If φ is a sentence then φ is also (the notation for) a term of the meta-logic. Thus we take $(\cdot)^*$ to be the identity map on notation. To see that this has the correct properties one proves, by an easy induction on the structure of terms and formulae, that if all free variables in t and φ are contained in the finite set X of variables, then $\{x\!:\!\iota \mid x \in X\} \rhd_{\Sigma_A} t\!:\!\iota$ and $\{x\!:\!\iota \mid x \in X\} \rhd_{\Sigma_A} \varphi\!:\!o$, and moreover t and φ are both in long-$\beta\eta$ normal form. Further, by induction on the structure of long-$\beta\eta$ normal forms of terms of type ι and o in contexts of the form $\{x\!:\!\iota \mid x \in X\}$, one shows that all such terms arise uniquely from a t or a φ respectively. Thus, specialising to the case $X = \emptyset$, we have that $(\cdot)^*$ is a bijection as required.

Next we should prove fullness. This step is easy in principle but tedious to carry out in detail, so we give only the general outline. Take any Hilbert system for first-order logic with equality, together with the axioms for Q (note that the Hilbert system chosen does not have to be the same as that implicitly given in $\mathcal{A}_Q$). We want to translate derivations of φ from assumptions $\varphi_1, \ldots, \varphi_n$ in the Hilbert system (where all the $\varphi, \varphi_1, \ldots, \varphi_n$ are sentences) onto proofs of $true(\varphi_1), \ldots, true(\varphi_n) \vdash_{(\Sigma_A, \mathcal{A}_Q)} true(\varphi)$. For this it is necessary to translate derivations of open formulae from open formulae. Suppose then that $\varphi, \varphi_1, \ldots, \varphi_n$ are formulae with all free variables contained in the finite set $X = \{x_1, \ldots, x_k\}$. We write $\forall X\!:\!\iota.\ \Phi$ for the meta-formula $\forall x_1\!:\!\iota. \ldots \forall x_k\!:\!\iota.\ \Phi$. One can now prove, by induction on derivations in the Hilbert system, that if φ follows from $\varphi_1, \ldots, \varphi_n$, then:

$$\forall X\!:\!\iota.\ true(\varphi_1), \ldots, \forall X\!:\!\iota.\ true(\varphi_n) \vdash_{(\Sigma_A, \mathcal{A}_Q)} \forall X\!:\!\iota.\ true(\varphi)$$

Again, specialising this to the case $X = \emptyset$ gives the desired result.

Only adequacy remains to be proved. To this end we construct a $(\Sigma_A, \mathcal{A}_Q)$-model satisfying the condition of Proposition 5.2. Let Mod_Q be the "set" of all classical models of Q. For each $\mathcal{M}$ in Mod_Q we distinguish its internal structure as $(N_\mathcal{M}, 0_\mathcal{M}, s_\mathcal{M}, +_\mathcal{M}, \cdot_\mathcal{M})$. We are going to construct a $(\Sigma_A, \mathcal{A}_Q)$-model $(W_Q, \leq_Q)$. The partially ordered set of worlds is given by $(W_Q, \leq_Q) = (\wp(Mod_Q), \supseteq)$ (the reader who is bothered by the size problem should make some suitable restriction such as only considering models, $\mathcal{M}$, with domain, $N_\mathcal{M}$, contained in a given infinite set). A world is thus a subset $\mathcal{S} \subseteq Mod_Q$. Before giving the interpretations of the different components of the Σ_A-structure at each world, we first interpret the type structure "pointwise" at each $\mathcal{M} \in Mod_Q$. The interpretation at $\mathcal{S}$ is then given by the $\mathcal{S}$-indexed direct product of the relevant pointwise interpretations.

Let $\mathcal{M}$ be any model of Q. The pointwise interpretation of types at $\mathcal{M}$ is given by the full type hierarchy:

$$\begin{array}{rcl} [\![\iota]\!]_{\mathcal{M}} & = & N_{\mathcal{M}} \\ [\![o]\!]_{\mathcal{M}} & = & \Omega \\ [\![A \to B]\!]_{\mathcal{M}} & = & [\![B]\!]_{\mathcal{M}}^{[\![A]\!]_{\mathcal{M}}} \end{array}$$

where $\Omega = \{\mathrm{tt}, \mathrm{ff}\}$, the set of truth values. Rather than give an exhaustive treatment of the constants in Σ_A, we define a representative sample. The interpretation of $\neg$, $\forall$, $=$ and $+$ at $\mathcal{M}$ will be elements $[\![\neg]\!]_{\mathcal{M}} \in \Omega^{\Omega}$, $[\![\forall]\!]_{\mathcal{M}} \in \Omega^{\Omega^{N_{\mathcal{M}}}}$, $[\![=]\!]_{\mathcal{M}} \in (\Omega^{N_{\mathcal{M}}})^{N_{\mathcal{M}}}$ and $[\![+]\!]_{\mathcal{M}} \in (N_{\mathcal{M}}^{N_{\mathcal{M}}})^{N_{\mathcal{M}}}$ respectively. These are defined by:

$$\begin{array}{rcl} [\![\neg]\!]_{\mathcal{M}}(b) & = & \begin{cases} \mathrm{tt} & \text{if } b = \mathrm{ff} \\ \mathrm{ff} & \text{if } b = \mathrm{tt} \end{cases} \\ [\![\forall]\!]_{\mathcal{M}}(f) & = & \begin{cases} \mathrm{tt} & \text{if, for all } a \in N_{\mathcal{M}},\ f(a) = \mathrm{tt} \\ \mathrm{ff} & \text{otherwise} \end{cases} \\ [\![=]\!]_{\mathcal{M}}(a_1)(a_2) & = & \begin{cases} \mathrm{tt} & \text{if } a_1 = a_2 \\ \mathrm{ff} & \text{otherwise} \end{cases} \\ [\![+]\!]_{\mathcal{M}}(a_1)(a_2) & = & +_{\mathcal{M}}(a_1, a_2) \end{array}$$

In short, the constants are interpreted in line with their real world meanings according to the Tarskian semantics of first-order logic.

The combinators K^{AB} and S^{ABC} are given the usual pointwise interpretations. For example, $K^{AB}_{\mathcal{M}} \in ([\![A]\!]_{\mathcal{M}}^{[\![B]\!]_{\mathcal{M}}})^{[\![A]\!]_{\mathcal{M}}}$ is defined by $K^{AB}_{\mathcal{M}}(a)(b) = a$.

The above interpretations of constants at $\mathcal{M}$ can be extended to all well-typed terms just by taking the usual interpretation of the typed lambda calculus in the full type hierarchy. Thus if $\Gamma \rhd_{\Sigma_A} M : A$ and γ is a environment function mapping each variable x_i with declaration $x_i : A_i$ in Γ to an element of $[\![A_i]\!]_{\mathcal{M}}$, then M has an interpretation $[\![M]\!]^{\gamma}_{\mathcal{M}} \in [\![A]\!]_{\mathcal{M}}$. When $\mathcal{M}$ is closed we just write $[\![M]\!]_{\mathcal{M}} \in [\![A]\!]_{\mathcal{M}}$. The crucial property of the pointwise interpretation, the mimicry of the Tarskian semantics, is given by the lemma below.

Lemma 5.3 *If $\rhd_{\Sigma_A} \varphi : o$ then $[\![\varphi]\!]_{\mathcal{M}} = tt$ if and only if $\mathcal{M} \models \varphi$.*

Proof. The proof is by induction on the structure of φ. The induction passes through open formulae and terms using the established correspondences between these and long-$\beta\eta$ normal forms of appropriate type.

For terms, suppose all variables in t are contained in the set $X = \{x_1, \ldots, x_k\}$. Given $\gamma : X \to N_{\mathcal{M}}$ (recall $[\![\iota]\!]_{\mathcal{M}} = N_{\mathcal{M}}$), one shows by an induction on t (similar to but easier than the induction on φ given below) that $[\![t]\!]^{\gamma}_{\mathcal{M}}$ is just the usual first-order interpretation of t in $\mathcal{M}$ according to the environment γ.

Now, for formulae, suppose all free variables in φ are contained in X as above. We show, by induction on φ, that, for all γ as above, $[\![\varphi]\!]^{\gamma}_{\mathcal{M}} = \mathrm{tt}$ if and only if $\mathcal{M} \models_{\gamma} \varphi$. The four cases are:

- $\boldsymbol{t_1 = t_2}$. $[\![t_1 = t_2]\!]^{\gamma}_{\mathcal{M}} = \mathrm{tt}$ if and only if (by definition of $[\![=]\!]_{\mathcal{M}}$) $[\![t_1]\!]^{\gamma}_{\mathcal{M}} = [\![t_2]\!]^{\gamma}_{\mathcal{M}}$ if and only if (by established result on terms) $\mathcal{M} \models_{\gamma} t_1 = t_2$.
- $\boldsymbol{\neg\varphi}$. $[\![\neg\varphi]\!]^{\gamma}_{\mathcal{M}} = \mathrm{tt}$ if and only if (by definition of $[\![\neg]\!]_{\mathcal{M}}$) $[\![\varphi]\!]^{\gamma}_{\mathcal{M}} = \mathrm{ff}$ if and only if (by induction hypothesis) $\mathcal{M} \not\models_{\gamma} \varphi$ if and only if $\mathcal{M} \models_{\gamma} \neg\varphi$.

$\varphi \Rightarrow \psi$. Similar to $\neg\varphi$.

$\forall x.\, \varphi$. $[\![\forall(\lambda X : \iota.\, \varphi)]\!]^{\gamma}_{\mathcal{M}} = \mathrm{tt}$ if and only if (by definition of $[\![\forall]\!]_{\mathcal{M}}$), for all $a \in N_{\mathcal{M}}$, $[\![\lambda X : \iota.\, \varphi]\!]^{\gamma}_{\mathcal{M}}(a) = \mathrm{tt}$ if and only if, for all $a \in N_{\mathcal{M}}$, $[\![\varphi]\!]^{\gamma[x:=a]}_{\mathcal{M}} = \mathrm{tt}$ (using an obvious notation for updating γ) if and only if (by induction hypothesis), for all $a \in N_{\mathcal{M}}$, $\mathcal{M} \models_{\gamma[x:=a]} \varphi$ if and only if $\mathcal{M} \models_{\gamma} \forall x.\, \varphi$.

The lemma follows by specialising to $X = \emptyset$. □

We can now define all the remaining components of the Σ_A-structure, $(W_Q, \leq_Q)$. Types, constants and combinators are interpreted at $\mathcal{S}$ by:

$$\begin{aligned} [\![A]\!]_{\mathcal{S}} &= \Pi_{\mathcal{M}\in\mathcal{S}}[\![A]\!]_{\mathcal{M}} \\ [\![c]\!]_{\mathcal{S}} &= \{[\![c]\!]_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}} \\ K^{AB}_{\mathcal{S}} &= \{K^{AB}_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}} \\ S^{ABC}_{\mathcal{S}} &= \{S^{ABC}_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}} \end{aligned}$$

The only predicate, *true*, is given the interpretation:

$$[\![\mathit{true}]\!]_{\mathcal{S}}(\{b_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}) \quad \text{iff} \quad \text{for all } \mathcal{M} \in \mathcal{S},\ b_{\mathcal{M}} = \mathrm{tt}$$

Finally, $\epsilon^{AB}_{\mathcal{S}}$ and $i^{A}_{\mathcal{S}\mathcal{S}'}$ (where $\mathcal{S}' \subseteq \mathcal{S}$) are defined by:

$$\begin{aligned} \epsilon^{AB}_{\mathcal{S}}(\{f_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}, \{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}) &= \{f_{\mathcal{M}}(a_{\mathcal{M}})\}_{\mathcal{M}\in\mathcal{S}} \\ i^{A}_{\mathcal{S}\mathcal{S}'}(\{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}) &= \{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}'} \end{aligned}$$

It is straightforward to check the required conditions showing that the construction above does indeed give a Σ_A-structure.

A basic but important property of the defined Σ_A-structure relates the interpretation of terms at $\mathcal{S}$ to their pointwise interpretations at all $\mathcal{M} \in \mathcal{S}$.

Lemma 5.4 *Suppose $\Gamma \rhd_{\Sigma_A} M : A$ and ρ interprets Γ at $\mathcal{S}$. Define, for $\mathcal{M} \in \mathcal{S}$, a function $\gamma^{\rho}_{\mathcal{M}} : X \to [\![A]\!]_{\mathcal{M}}$ (where X is the set of variables in Γ) by:*

$$\gamma^{\rho}_{\mathcal{M}}(x) \quad = \quad a_{\mathcal{M}} \textit{ where } \rho(x)_{\mathcal{S}} = \{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$$

Then $[\![M]\!]^{\rho}_{\mathcal{S}} = \{[\![M]\!]^{\gamma^{\rho}_{\mathcal{M}}}_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$.

Proof. By a straightforward induction on the structure on M. □

Finally we are in a position to establish the facts we need to prove the faithfulness of (Σ_A, A_Q). In the results below we make frequent use of the notation introduced on page 323.

Lemma 5.5 *$\mathit{form}(\mathcal{S}) = \mathit{Th}_{\models}(\mathcal{S})$.*

Proof. We must show that $\mathcal{S} \models \mathit{true}(\varphi)$ if and only if, for all $\mathcal{M} \in \mathcal{S}$, $\mathcal{M} \models \varphi$. But $\mathcal{S} \models \mathit{true}(\varphi)$ if and only if $[\![\varphi]\!]_{\mathcal{S}} = \{tt\}_{\mathcal{M}\in\mathcal{S}}$ if and only if (by Lemma 5.4), for all $\mathcal{M} \in \mathcal{S}$, $[\![\varphi]\!]_{\mathcal{M}} = \mathrm{tt}$ if and only if (by Lemma 5.3), for all $\mathcal{M} \in \mathcal{S}$, $\mathcal{M} \models \varphi$. □

Proposition 5.6 *$(W_Q, \leq_Q)$ is a $(\Sigma_A, \mathcal{A}_Q)$-model.*

Proof. We must show that, for all $\Phi \in \mathcal{A}_Q$, $(W_Q, \leq_Q) \models \Phi$. First note that for the last seven axioms of $\mathcal{A}_Q$ in Figure 7 this follows immediately from Lemma 5.5. Of the other eight meta-axioms we validate three.

1. $\forall \varphi{:}o.\ \forall \psi{:}o.\ \mathit{true}(\varphi \Rightarrow \psi \Rightarrow \varphi)$.

   Suppose we have $\mathcal{S}$ and ρ with $\rho(\varphi)_{\mathcal{S}} = \{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$ and $\rho(\psi)_{\mathcal{S}} = \{b_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$. Then:

$$\begin{array}{rcll} [\![\varphi \Rightarrow \psi \Rightarrow \varphi]\!]^{\rho}_{\mathcal{S}} & = & \{[\![\Rightarrow]\!]_{\mathcal{M}}(a_{\mathcal{M}})([\![\Rightarrow]\!]_{\mathcal{M}}(b_{\mathcal{M}})(a_{\mathcal{M}}))\}_{\mathcal{M}\in\mathcal{S}} & \text{(by Lemma 5.4)} \\ & = & \{\mathrm{tt}\}_{\mathcal{M}\in\mathcal{S}} & \end{array}$$

   So $\mathcal{S} \models_{\rho} \mathit{true}(\varphi \Rightarrow \psi \Rightarrow \varphi)$ as required.

2. $\forall \varphi{:}o.\ \forall \psi{:}\iota \to o.\ (\forall x{:}\iota.\ \mathit{true}(\varphi \Rightarrow \psi(x))) \supset \mathit{true}(\varphi \Rightarrow \forall x.\ \psi(x))$.

   Suppose we have $\mathcal{S}$ and ρ with $\rho(\varphi)_{\mathcal{S}} = \{b_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$ and $\rho(\psi)_{\mathcal{S}} = \{f_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$ such that $\mathcal{S} \models_{\rho} \forall x : \iota.\ \mathit{true}(\varphi \Rightarrow \psi(x))$. Then for all $\{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}} \in [\![\iota]\!]_{\mathcal{S}}$, $\mathcal{S} \models_{\rho[x:=\{a_{\mathcal{M}}\}]} \mathit{true}(\varphi \Rightarrow \psi(x))$. So, by Lemma 5.4, for all $a_{\mathcal{M}} \in N_{\mathcal{M}}$, $[\![\Rightarrow]\!]_{\mathcal{M}}(b_{\mathcal{M}})(f_{\mathcal{M}}(a_{\mathcal{M}})) = \mathrm{tt}$. But then $[\![\Rightarrow]\!]_{\mathcal{M}}(b_{\mathcal{M}})([\![\forall]\!]_{\mathcal{M}}(f_{\mathcal{M}})) = \mathrm{tt}$ (by the usual reasoning). And thus, again by Lemma 5.4, $[\![\varphi \Rightarrow \forall x.\ \psi(x))]\!]^{\rho}_{\mathcal{S}} = \{\mathrm{tt}\}_{\mathcal{M}\in\mathcal{S}}$. So $\mathcal{S} \models_{\rho} \mathit{true}(\varphi \Rightarrow \forall x.\ \psi(x))$ as required.

3. $\forall \varphi{:}\iota \to o.\ \forall t_1{:}\iota.\ \forall t_2{:}\iota.\ \mathit{true}(t_1 = t_2) \supset \mathit{true}(\varphi(t_1)) \supset \mathit{true}(\varphi(t_2))$.

   Suppose we have $\mathcal{S}$ and ρ with $\rho(\varphi)_{\mathcal{S}} = \{f_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$, $\rho(t_1)_{\mathcal{S}} = \{a_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$ and $\rho(t_2)_{\mathcal{S}} = \{b_{\mathcal{M}}\}_{\mathcal{M}\in\mathcal{S}}$ such that $\mathcal{S} \models_{\rho} \mathit{true}(t_1 = t_2)$ and $\mathcal{S} \models_{\rho} \mathit{true}(\varphi(t_1))$. Then, applying Lemma 5.4 as above, it is easy to see that, for all $\mathcal{M} \in \mathcal{S}$, $a_{\mathcal{M}} = b_{\mathcal{M}}$ and $f_{\mathcal{M}}(a_{\mathcal{M}}) = \mathrm{tt}$, so obviously $f_{\mathcal{M}}(b_{\mathcal{M}}) = \mathrm{tt}$. Now, again by Lemma 5.4, $\mathcal{S} \models_{\rho} \mathit{true}(\varphi(t_2))$ as required.

The other axioms are validated in a similar way. □

The faithfulness of $(\Sigma_A, \mathcal{A}_Q)$ now follows easily. We need only show that $(W_Q, \leq_Q)$ has the property required by Proposition 5.2. Let, T, be a theory of $\vdash_Q$. We must show that $T = \mathit{form}(\mathcal{S})$ for some $\mathcal{S} \in W_Q$. Define $\mathcal{S}_T = \mathit{Mods}_{\models}(T)$. By the completeness theorem for first-order logic, $\varphi \in T$ if and only if, for all $\mathcal{M} \in \mathcal{S}_T$, $\mathcal{M} \models \varphi$. So by Lemma 5.5, $T = \mathit{form}(\mathcal{S}_T)$. Note that the reader who has taken W_Q to be defined over models, $\mathcal{M}$, with $N_{\mathcal{M}}$ contained in a given infinite set must do a little more work. In this case, $\mathcal{S}_T$ is defined to be the set of models of T of the appropriate form, so an appropriate weak form of the Löwenheim-Skolem theorem is required to give completeness in terms of the models considered.

Perhaps the above proof of adequacy seems rather long-winded. Indeed, a syntactic proof along the lines of those in [6, 12] would have probably been shorter. However, now that we have constructed the model and established its properties we are free to exploit it without repeating the effort. This we will do on page 338 to establish, very easily, the adequacy of an extension of $(\Sigma_A, \mathcal{A}_Q)$. Another benefit of the semantic proof is that the model constructed gives us some intuition as to the meaning of meta-logical propositions.

Clearly, the above proof does not rely on any properties particular to Robinson's Q. In fact analogous methods work for any first-order theory. However, now we confess to why we considered Q in preference to Peano Arithmetic (PA). One would like to axiomatise the induction schema by a single meta-axiom:

$$\forall \varphi{:}\iota \to o.\ \mathit{true}(\varphi(0) \Rightarrow (\forall x.\ \varphi(x) \Rightarrow \varphi(s(x))) \Rightarrow \forall x.\varphi(x))$$

And indeed adding this axiom to $\mathcal{A}_Q$ (we call the new set $\mathcal{A}_{PA}$) does give an adequate presentation of PA. Unfortately, the Σ_A-structure constructed, following the above method, out of models of PA is not a $(\Sigma_A, \mathcal{A}_{PA})$-model. The problem is easy to identify. The axiom giving the induction principle has the full power of the second-order induction axiom in the Σ_A-structure. Thus it can not hold at any world (sets of models of PA) that contains a non-standard model of arithmetic. One might try to patch this by restricting the pointwise interpretations of predicate types to arithmetically definable relations. Thus $[\![\iota \to o]\!]_{\mathcal{M}}$ and $[\![\iota \to \iota \to o]\!]_{\mathcal{M}}$ would be respectively the unary and binary arithmetic relations in $\mathcal{M}$. Unfortunately, this attempted remedy is also doomed. Semantically we must be able to apply any element of $[\![\iota \to \iota \to o]\!]_{\mathcal{M}}$ to any element of $[\![\iota]\!]_{\mathcal{M}}$ and obtain an element of $[\![\iota \to o]\!]_{\mathcal{M}}$. But if we apply a binary arithmetic relation to a non-standard element of $N_{\mathcal{M}}$, the resulting unary relation is no longer arithmetically definable.

One way of avoiding the problem is to take, in place of the above meta-axiom for induction, all instances of the following meta-schema:

$$\mathit{true}(\varphi(0) \Rightarrow (\forall x.\ \varphi(x) \Rightarrow \varphi(s(x))) \Rightarrow \forall x. \varphi(x))$$

(where φ ranges over $\{\varphi \mid \rhd_{\Sigma_A} \varphi : \iota \to o,\ \varphi \text{ is in long-}\beta\eta \text{ normal form}\}$). With this schema the constructed Σ_A-structure does indeed give a model. However, this is clearly an unsatisfactory solution. The presentation is infinite whereas $(\Sigma_A, \mathcal{A}_{PA})$ is an finite presentation of PA.

It is possible to prove the adequacy of $(\Sigma_A, \mathcal{A}_{PA})$ by constructing a syntactic model over the lattice of theories of PA. We do not give details, as there is an example of a of similar (but simpler) construction in the next section. A syntactic model could also be used to prove the faithfulness of $(\Sigma_A, \mathcal{A}_Q)$. However, we preferred to give the above proof to show how, in some cases, an entirely semantic proof is possible.

6 Admissible rules

In this section we develop a simple theory of sequent rules for an arbitrary logic. Given an encoding of the logic, each rule has a corresponding meta-formula expressing its admissibility. We use the semantics to investigate under what conditions the meta-formulae corresponding to admissible rules are theorems in the meta-theory given by an adequate presentation.

Throughout this section $(\mathcal{L}, \vdash)$ is an arbitrary consequence relation. When referred to, $(\Sigma, \mathcal{A})$ is assumed to be a presentation giving, by means of the mapping $(\cdot)^*$, an adequate encoding of $(\mathcal{L}, \vdash)$.

In examples we will often assume that each $\mathcal{L}$ has a binary implication connective, $\Rightarrow$. By this we mean that there is a distinguished injective function from $\mathcal{L} \times \mathcal{L}$ to $\mathcal{L}$, and we write $\varphi \Rightarrow \psi$ for the value of this function when applied to φ and ψ. We say that $\vdash$ satisfies *modus ponens* if:

$$\text{for all } \Delta,\ \varphi \text{ and } \psi, \text{ if } \Delta \vdash \varphi \Rightarrow \psi \text{ and } \Delta \vdash \varphi \text{ then } \Delta \vdash \psi.$$

We say that $\vdash$ satisfies the *deduction theorem* if:

$$\text{for all } \Delta,\ \varphi \text{ and } \psi, \text{ if } \Delta, \varphi \vdash \psi \text{ then } \Delta \vdash \varphi \Rightarrow \psi.$$

A *sequent* is a non-empty finite sequence of formulae $(\varphi_1, \ldots, \varphi_m, \varphi)$. Normally one would write $\varphi_1, \ldots, \varphi_m \vdash \varphi$. The reason for using such a neutral notation is that the symbols "$\vdash$", "$\to$", "$\supset$" and "$\Rightarrow$" are already tied up. A *rule* has the form:

$$\frac{S_1 \quad \ldots \quad S_n}{S} \tag{3}$$

where $n \geq 0$ and $S_1, \ldots, S_n, S$ are all sequents. A sequent, $(\varphi_1, \ldots, \varphi_m, \varphi)$, is said to be Δ-*valid* if $\Delta, \varphi_1, \ldots, \varphi_m \vdash \varphi$. A rule (of the above form) is said to be *admissible* if:

for all Δ, if, for all i $(1 \leq i \leq n)$, S_i is Δ-valid then S is Δ-valid.

Thus, for example, the rule:

$$\frac{(\varphi, \psi)}{(\varphi \Rightarrow \psi)} \tag{4}$$

is admissible for all φ and ψ if and only if the deduction theorem holds for $\vdash$.

Given a sequent $S = (\varphi_1, \ldots, \varphi_m, \varphi)$, consider the meta-formula, which we call S^*, below.

$$\mathit{true}(\varphi_1^*) \supset \ldots \supset \mathit{true}(\varphi_m^*) \supset \mathit{true}(\varphi^*)$$

Intuitively this meta-formula expresses the meta-proposition that $\varphi_1, \ldots, \varphi_m \vdash \varphi$ holds. Now given a rule, $\mathcal{R}$, of the general form (3), define its translation, $\mathcal{R}^*$, to be the meta-formula below.

$$S_1^* \supset \ldots \supset S_n^* \supset S^*$$

Intuitively, this meta-formula expresses that the rule is admissible. The above intuitions will be given substance by results below.

It is worth commenting on our notions of rule and admissibility and how these relate to the usual notions. The most common notion of admissible rule is that given by Troelstra [16, §1.11.1] (see also [7, pages 69–70]). However, there the notion of rule is one between formulae rather than sequents and the notion of admissibility is with respect to a fixed theory rather than a consequence relation. For us the consequence relation is of primary interest. It is for this reason that we consider a more general form of rule with sequents for premisses and conclusion. Now the notion of admissibility for a rule of the form (3) depends on how we stipulate that this rule is to be applied. Suppose $\mathcal{S}_i = (\varphi_{i1}, \ldots, \varphi_{im_i}, \psi_i)$ $(1 \leq i \leq m)$ and $\mathcal{S} = (\varphi_1, \ldots, \varphi_m, \varphi)$. One possibility would be to stipulate that the rule is only applicable when, for all $1 \leq i \leq n$, $\varphi_{i1}, \ldots, \varphi_{im_i} \vdash \psi_i$; in which case one deduces that $\varphi_1, \ldots, \varphi_m \vdash \varphi$. Clearly this notion of rule application induces a different notion of admissibility from ours. Our definition of admissibility is motivated instead by the natural-deduction conception of rule application. Written in a more suggestive format, we think of the rule above as:

$$\frac{\begin{array}{c}[\varphi_{11}, \ldots, \varphi_{1m_1}]\\ \psi_1\end{array} \quad \cdots \quad \begin{array}{c}[\varphi_{n1}, \ldots, \varphi_{nm_n}]\\ \psi_n\end{array} \qquad \varphi_1 \quad \cdots \quad \varphi_m}{\varphi}$$

So the rule is applicable if, for all $1 \leq i \leq n$, $\Delta, \varphi_{i1}, \ldots, \varphi_{im_i} \vdash \psi_i$ and, for all $1 \leq i \leq m$, $\Delta \vdash \varphi_i$; in which case we deduce $\Delta \vdash \varphi$, the formulae in Δ being carried around as unused assumptions. With a little shuffling of the conseqences it is easy to see that our definition of admissibility is now the appropriate one. We remark here that it would be interesting to consider notions of admissibility for the more general higher-order rules of Schroeder-Heister [13]. However, the (second-order) rules we consider include all the natural deduction rules in common usage.

One serious defect in our notion of rule is that each rule is considered only with respect to fixed formulae. Usually rules are schematic. The deduction theorem would normally be considered as just one rule in the shape of (4) schematic over φ and ψ. One would then prefer the following translation (which we henceforth call DT):

$$\forall \varphi{:}o.\, \forall \psi{:}o.\, (\mathit{true}(\varphi) \supset \mathit{true}(\psi)) \supset \mathit{true}(\varphi \Rightarrow \psi)$$

The problem with this approach is that, in order to make sense of a general notion of schematic rule, we require some theory of propositional connectives in $\mathcal{L}$. Similarly, for DT to be well

$$
\begin{array}{rcl}
\mathcal{T}_m & = & \{o\} \\
\mathcal{P}_m & = & \{true\!:\!\langle o\rangle\} \\
\Sigma_m & = & \{\Rightarrow\!: o \to o \to o,\ a : o,\ b : o\} \\
\mathcal{A}_m & = & \{\forall\varphi\!:\!o.\ \forall\psi\!:\!o.\ true(\varphi \Rightarrow \psi \Rightarrow \varphi), \\
& & \quad \forall\varphi\!:\!o.\ \forall\psi\!:\!o.\ \forall\theta\!:\!o.\ true((\varphi \Rightarrow \psi \Rightarrow \theta) \Rightarrow (\varphi \Rightarrow \psi) \Rightarrow \varphi \Rightarrow \theta), \\
& & \quad \forall\varphi\!:\!o.\ \forall\psi\!:\!o.\ true(\varphi \Rightarrow \psi) \supset true(\varphi) \supset true(\psi)\} \\
\mathcal{A}'_m & = & \mathcal{A}_m \cup \{((true(a) \supset true(b)) \supset true(a)) \supset true(a)\}
\end{array}
$$

Figure 8: Presentations of propositional minimal implicational logic.

formed, we require a consant $\Rightarrow\!: o \to o \to o$ in Σ. Further, there would have to be extra conditions on the translation $(\cdot)^*$ ensuring preservation of connectives and compositionality. These are perfectly reasonable requirements (see Σ_A for example), but would take us away from the simple-minded notion of logic of section 4. However, when possible, we shall consider DT in the examples that follow.

A first and simple connection between the adequacy of rules and the theoremhood of their meta-logical translations is given by the proposition below. Remember that $(\Sigma, \mathcal{A})$ is assumed to give an adequate encoding of $(\mathcal{L}, \vdash)$.

Lemma 6.1 *For all sequents, S, S is Δ-valid if and only if $true(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} S^*$.*

Proof. A sequent, $S = (\varphi_1, \ldots, \varphi_m, \varphi)$, is Δ-valid if and only if $\Delta, \varphi_1, \ldots, \varphi_m \vdash \varphi$ if and only if (by adequacy) $true(\Delta^*), true(\varphi_1^*), \ldots, true(\varphi_n^*) \vdash_{(\Sigma,\mathcal{A})} true(\varphi^*)$ if and only if $true(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} S^*$. □

Proposition 6.2 *For all rules $\mathcal{R}$, if $\vdash_{(\Sigma,\mathcal{A})} \mathcal{R}^*$ then $\mathcal{R}$ is admissible.*

Proof. Suppose $\vdash_{(\Sigma,\mathcal{A})} S_1^* \supset \ldots \supset S_n^* \supset S^*$. Take an arbitrary Δ such that, for all i $(1 \leq i \leq n)$, S_i is Δ-valid. We must show that S is Δ-valid. Now, by the lemma, for all i $(1 \leq i \leq n)$, $true(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} S_i^*$. So clearly $true(\Delta^*) \vdash_{(\Sigma,\mathcal{A})} S^*$. But then, again by the lemma, S is Δ-valid as required. □

In [6] it is noted that the meta-logic has no facility for induction, so there is no general way of proving the admissibility of inference rules. However, we are only considering instances of rules rather than quantified schematic rules, so induction can play no part. Nevertheless, as we prove below, the converse of Proposition 6.2 still fails: in general there are admissible rules $\mathcal{R}$ such that $\not\vdash_{(\Sigma,\mathcal{A})} \mathcal{R}^*$. It is therefore natural to consider extensions of $\mathcal{A}$ with unprovable translations of admissible rules. Define $Adm_\vdash = \{\mathcal{R}^* \mid \mathcal{R} \text{ is an admissible rule}\}$. Let R be a subset of $Adm_\vdash$. We say that $(\Sigma, \mathcal{A})$ is *safe for* R if $(\Sigma, \mathcal{A} \cup R)$ is adequate. Given that $(\Sigma, \mathcal{A})$ is adequate and, for each $\mathcal{R}^* \in R$, $\mathcal{R}$ is admissible, it might be imagined that $(\Sigma, \mathcal{A})$ is always safe for R. However, this is not the case. It is possible that the addition of the translation of an admissible rule to an adequate signature can result in the loss of adequacy! We now develop an example to show all this.

We first show that the presentation, $(\mathcal{T}_m, \mathcal{P}_m, \Sigma_m, \mathcal{A}'_m)$, in Figure 8 gives an adequate presentation of minimal implicational logic over two propositional constants, $(\mathcal{L}_m, \vdash_m)$. The formulae, φ, in $\mathcal{L}_m$ are given by the grammar:

$$\varphi \ ::= \ a \mid b \mid \varphi \Rightarrow \psi$$

The consequence relation, $\vdash_m$, is just intuitionistic entailment over $\mathcal{L}_m$.

The mapping, $(\cdot)^*$, from formulae to long-$\beta\eta$ normal forms of type o is evident (again it is the identity map on notation so we omit the * appendage). Bijectivity is proved by an easy induction on the structure of long-$\beta\eta$ normal forms. Similarly, there is an evident mapping from proofs in the usual Hilbert system for $\vdash_m$ (see, for example, [7, page 193]) of $\varphi_1, \ldots, \varphi_n \vdash \varphi$ to proofs of $true(\varphi_1), \ldots, true(\varphi_n) \vdash_{(\Sigma_m, \mathcal{A}_m)} true(\varphi)$, so the fullness of $(\Sigma_m, \mathcal{A}_m)$ is easy to establish. The fullness of $(\Sigma_m, \mathcal{A}'_m)$ follows (as $\mathcal{A}_m \subseteq \mathcal{A}'_m$).

To prove faithfulness we construct a suitable $(\Sigma_m, \mathcal{A}'_m)$-model. Although such a model could be constructed out of Kripke models of $\vdash_m$, we construct instead a model of the syntactic kind alluded to in the discussion at the end of section 5. The model is defined over the discrete partial order $(W_m, =)$ where $W_m = \mathit{Theories}_{\vdash_m}$. The type structure is the same at each world, and is given by the extensional collapse of the closed terms of Σ_m. Thus the interpretation of A is the quotient of $\{M \mid \rhd_{\Sigma_m} M : A\}$ by an equivalence relation $\sim_A$, where $\sim_A$ is inductively defined by setting $\sim_o$ to $=_{\beta\eta}$ and defining:

$$M \sim_{A \to B} M' \quad \text{iff} \quad \text{for all } N,\, N', \text{ if } N \sim_A N' \text{ then } M(N) \sim_B M'(N')$$

It is easy to show (by induction on types) that $\sim_A$ is a partial equivalence relation. It is also easy to see that $c \sim_A c$ for the three constants in Σ_m, as they are at worst first-order. Now the reflexivity of $\sim_A$, for all A, follows from the well-known "Basic Lemma" of logical relations (see [9, §3]).

The entire Σ_m-structure is defined as follows.

$$\begin{array}{rcl}
[\![A]\!]_T & = & \{M \mid \rhd_{\Sigma_m} M : A\}/\sim_A \\
[\![true]\!]_T([M]) & \text{iff} & \beta\eta(M) = \varphi^* \text{ for some } \varphi \in T \\
[\![c]\!]_T & = & [c] \\
\epsilon_T^{AB}([M],[N]) & = & [M(N)] \\
K_T^{AB} & = & [\lambda x : A.\, \lambda y : B.\, x] \\
S_T^{ABC} & = & [\lambda x : A \to B \to C.\, \lambda y : A \to B.\, \lambda z : A.\, x(z)(y(z))]
\end{array}$$

There is no need to specify $i^A_{TT'}$ as the partial order is discrete, so $i^A_{TT'}$ is only defined for $T = T'$ in which case it must be the identity. Of the various conditions that have to be satisfied for $(W_m, =)$ to be a Σ_m-structure, the most interesting is extensionality. However, this is easily shown by induction on the structure of types. The other conditions are trivially verified.

It is now possible to prove the adequacy of $(\Sigma_m, \mathcal{A}'_m)$.

Proposition 6.3 *$(W_m, =)$ is a $(\Sigma_m, \mathcal{A}'_m)$-model.*

Proof. We must show that, for all $\Phi \in \mathcal{A}'_m$, $(W_Q, \leq_Q) \models \Phi$.

The three axioms in $\mathcal{A}_m$ are all validated easily. We consider only the third:

$$\forall \varphi : o.\, \forall \psi : o.\; true(\varphi \Rightarrow \psi) \supset true(\varphi) \supset true(\psi)$$

Suppose we have T and ρ with $\rho(\varphi)_T = [M]$, $\rho(\psi)_T = [N]$ such that $T \models_\rho true(\varphi \Rightarrow \psi)$ and $T \models_\rho true(\varphi)$. But $\beta\eta(M)$ and $\beta\eta(N)$ must correspond to formulae of $\mathcal{L}_m$, φ and ψ say (not to be confused with the meta-variables). But then $\varphi \Rightarrow \psi \in T$ and $\varphi \in T$, so by modus ponens $\psi \in T$. Thus clearly $T \models_\rho true(\psi)$ as required.

For the fourth axiom:

$$((\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a)) \supset \mathit{true}(a)$$

Suppose we have T such that $T \models (\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a)$. Now suppose for contradiction that $T \not\models \mathit{true}(a)$. Then, as the partial order is discrete, it is clear that $T \models \mathit{true}(a) \supset \mathit{true}(b)$. So $T \models \mathit{true}(a)$, giving the desired contradiction. □

In this case the property required by Proposition 5.2 is evident. Given $T \in \mathit{Theories}_{\vdash_m}$, it is immediate from the construction of the model that $\mathit{form}(T) = T$. So $(\Sigma_m, \mathcal{A}'_m)$ is indeed faithful and hence adequate. Note that we have also established the adequacy of $(\Sigma_m, \mathcal{A}_m)$.

We now address the issues raised above, providing a counterexample to the converse of Proposition 6.2 and showing that the stated problem with safety does indeed arise. It is well-known that $\vdash_m$ satisfies the deduction theorem. So setting:

$$\begin{aligned} R &= \{(\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a \Rightarrow b), \\ &\quad (\mathit{true}((a \Rightarrow b) \Rightarrow a) \supset \mathit{true}(a)) \supset \mathit{true}(((a \Rightarrow b) \Rightarrow a) \Rightarrow a)\} \end{aligned}$$

we have that $R \subseteq \mathit{Adm}_{\vdash_m}$. Either of the meta-formulae in R provides a counterexample to the converse of Proposition 6.2. We just show one case.

Proposition 6.4 $\not\vdash_{(\Sigma_m, \mathcal{A}'_m)} (\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a \Rightarrow b)$.

Proof. By soundness of the meta-logic it is enough to find a T such that $T \not\models (\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a \Rightarrow b)$ in $(W_m, =)$. Define $T = \mathit{Th}_{\vdash_m}(\emptyset)$. Then clearly $T \models \mathit{true}(\varphi)$ if and only if φ is a theorem of minimal logic. Thus $T \not\models \mathit{true}(a)$ so, as the partial order is discrete, $T \models \mathit{true}(a) \supset \mathit{true}(b)$. But also $T \not\models \mathit{true}(a \Rightarrow b)$, so indeed $T \not\models (\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a \Rightarrow b)$. □

The above proposition was also observed by Randy Pollack who proved it syntactically.[1] The problem with safety is illustrated by the fact that $(\Sigma_m, \mathcal{A}'_m \cup R)$ is not adequate (hence, incidentally, neither is $(\Sigma_m, \mathcal{A}'_m \cup \{\mathrm{DT}\})$). It is easy to show that $\vdash_{(\Sigma_m, \mathcal{A}'_m \cup R)} \mathit{true}(((a \Rightarrow b) \Rightarrow a) \Rightarrow a)$, but it is well-known that $\not\vdash_m ((a \Rightarrow b) \Rightarrow a) \Rightarrow a$. Therefore $(\Sigma_m, \mathcal{A}'_m)$, though adequate, is not safe for R above.

In the above example, the source of the problem with safety is the axiom:

$$((\mathit{true}(a) \supset \mathit{true}(b)) \supset \mathit{true}(a)) \supset \mathit{true}(a) \tag{5}$$

This is redundant: $(\Sigma_m, \mathcal{A}_m)$ (which does not contain the axiom) is also adequate. Intuitively, (5) gives meta-implication an aspect of classical behaviour (it is an instance of Peirce's Law). $(\Sigma_m, \mathcal{A}'_m)$ is adequate as there is no need for object-level implication to behave in the same way as meta-implication. However, when R is added the two implications are forced to behave alike in two critical cases, and thus object implication inherits unwanted classical properties.

In view of the problem with safety, it is natural to consider conditions under which $(\Sigma, \mathcal{A})$ is safe for an arbitrary $R \subseteq \mathit{Adm}_\vdash$. Clearly, this is so if and only if $(\Sigma, \mathcal{A})$ is safe for $\mathit{Adm}_\vdash$ itself. A semantic condition for $(\Sigma, \mathcal{A})$ to be safe for $\mathit{Adm}_\vdash$ is given by Corollary 6.7 below.

Lemma 6.5 *If $(W, \leq)$ is a Σ-structure with $\mathit{form}(W) \subseteq \mathit{Theories}_\vdash$ such that, for all $T \supseteq \mathit{form}(w)$, there exists $w' \geq w$ with $\mathit{form}(w') = T$ then, for all sequents $\mathcal{S}$, for all $w \in W$, $w \models \mathcal{S}^*$ if and only if $\mathcal{S}$ is $\mathit{form}(w)$-valid.*

[1] Private communication.

Proof. Let $(W, \leq)$ be any Σ-structure satisfying the stated condition. Take any sequent $\mathcal{S} = (\varphi_1, \ldots, \varphi_m, \varphi)$ and any $w \in W$.

Suppose $w \models \mathcal{S}^*$, ie. $w \models \mathit{true}(\varphi_1^*) \supset \ldots \supset \mathit{true}(\varphi_m^*) \supset \mathit{true}(\varphi^*)$. Let $w' \geq w$ be such that $\mathit{form}(w') = \mathit{Th}_\vdash(\mathit{form}(w) \cup \{\varphi_1, \ldots, \varphi_m\})$ (such a w' is guaranteed to exist by the condition on $(W, \leq)$). Then $w' \models \mathit{true}(\varphi_i^*)$ $(1 \leq i \leq m)$, so $w' \models \mathit{true}(\varphi^*)$. But then $\varphi \in \mathit{form}(w')$ so $\mathit{form}(w), \varphi_1, \ldots, \varphi_m \vdash \varphi$ (by required property of w'). Thus indeed $\mathcal{S}$ is $\mathit{form}(w)$-valid.

Conversely, suppose $\mathcal{S}$ is $\mathit{form}(w)$-valid, ie. $\mathit{form}(w), \varphi_1, \ldots, \varphi_m \vdash \varphi$. Now suppose $w' \geq w$ is such that $w' \models \mathit{true}(\varphi_i^*)$ $(1 \leq i \leq m)$. Then $\mathit{form}(w) \cup \{\varphi_1, \ldots, \varphi_m\} \subseteq \mathit{form}(w')$. But $\mathit{form}(w') \in \mathit{Theories}_\vdash$ (by the condition on $(W, \leq)$), so $\varphi \in \mathit{form}(w')$. Thus $w' \models \mathit{true}(\varphi^*)$. This shows that $w \models \mathit{true}(\varphi_1^*) \supset \ldots \supset \mathit{true}(\varphi_m^*) \supset \mathit{true}(\varphi^*)$ as required. $\Box$

Proposition 6.6 *If $(W, \leq)$ is a Σ-structure with $\mathit{form}(W) = \mathit{Theories}_\vdash$ such that, for all $T \supseteq \mathit{form}(w)$, there exists $w' \geq w$ with $\mathit{form}(w') = T$ then for all rules, $\mathcal{R}$, $(W, \leq) \models \mathcal{R}^*$ if and only if $\mathcal{R}$ is admissible.*

Proof. Let $(W, \leq)$ be any Σ-structure satisfying the stated condition. Let $\mathcal{R}$ be any rule of the general form (3).

Suppose $(W, \leq) \models \mathcal{R}^*$, ie., for all $w \in W$, $w \models \mathcal{S}_1^* \supset \ldots \supset \mathcal{S}_n^* \supset \mathcal{S}^*$. Suppose further that all $\mathcal{S}_i$ $(1 \leq i \leq n)$ are Δ-valid. Then clearly all the $\mathcal{S}_i$ are $\mathit{Th}_\vdash(\Delta)$-valid. Let w be any world such that $\mathit{form}(w) = \mathit{Th}_\vdash(\Delta)$ (the existence of w is guaranteed by the condition on $(W, \leq)$). Then, by the lemma above, $w \models \mathcal{S}_i^*$ $(1 \leq i \leq n)$ so clearly $w \models \mathcal{S}^*$. But then, again by the lemma, $\mathcal{S}$ is $\mathit{Th}_\vdash(\Delta)$-valid and hence Δ-valid. Thus $\mathcal{R}$ is indeed admissible.

Conversely, suppose $\mathcal{R}$ is admissible. Take any $w \in W$ such that $w \models \mathcal{S}_i^*$ $(1 \leq i \leq n)$. Then, by the lemma, all $\mathcal{S}_i$ are $\mathit{form}(w)$-valid. So, by admissibility, $\mathcal{S}$ is $\mathit{form}(w)$-valid. But then, again by the lemma, $w \models \mathcal{S}^*$. So, for all $w \in W$, $w \models S_1^* \supset \ldots \supset S_n^* \supset S^*$ as required. $\Box$

Corollary 6.7 *If $(W, \leq)$ is a $(\Sigma, \mathcal{A})$-model with $\mathit{form}(W) = \mathit{Theories}_\vdash$ such that, for all $T \supseteq \mathit{form}(w)$, there exists $w' \geq w$ with $\mathit{form}(w') = T$ then $(\Sigma, \mathcal{A})$ is safe for $\mathit{Adm}_\vdash$.*

Proof. By the proposition $(W, \leq)$ is a $(\Sigma, \mathcal{A} \cup \mathit{Adm}_\vdash)$-model. So by Proposition 5.2, $(\Sigma, \mathcal{A} \cup \mathit{Adm}_\vdash)$ is faithful and hence adequate. $\Box$

Note that by propositions 5.1 and 5.2 we can always find a $(\Sigma, \mathcal{A})$-model with $\mathit{form}(W) = \mathit{Theories}_\vdash$. Thus the only extra condition to satisfy is that on the relationship between $\leq$ and $\subseteq$. Intuitively, the only property of a world, w, in which we are interested is the contents of $\mathit{form}(w)$. The condition establishes that *form* is a "zig-zag morphism" (see [17]) from $(W, \leq)$ to the lattice $(\mathit{Theories}_\vdash, \subseteq)$.

We can now show that $(\Sigma_m, \mathcal{A}_m)$ is safe for $\mathit{Adm}_{\vdash_m}$ and thus, unsurprisingly, better behaved than $(\Sigma_m, \mathcal{A}'_m)$. To this end we construct a $(\Sigma_m, \mathcal{A}_m)$-model satisfying the condition of Corollary 6.7. This time the model is built over the partial order $(W_m, \subseteq)$ (where again $W_m = \mathit{Theories}_{\vdash_m}$). All the components of the model are exactly as in the discrete case above. This time it is necessary to define $i^A_{TT'}$ (for $T \subseteq T'$), but this is simply:

$$i^A_{TT'}([M]) \quad = \quad [M]$$

It is readily checked that the resulting structure is a a $(\Sigma_m, \mathcal{A}_m)$-model (the proof goes through exactly as before). It is also easy to see that the conditions of Corollary 6.7 are satisfied as *form* is the identity morphism from the lattice $(\mathit{Theories}_{\vdash_m}, \subseteq)$ to itself. So indeed $(\Sigma_m, \mathcal{A}_m)$ is safe for $\mathit{Adm}_{\vdash_m}$.

To investigate the applicability of the theory developed in this section we return to the encoding $(\Sigma_A, \mathcal{A}_Q)$ of Robinson's Q given in the previous section. First we show that there are admissible rules not provable in $(\Sigma_A, \mathcal{A}_Q)$. It is well-known that the deduction theorem holds for classical logic, and hence for $\vdash_Q$ (remember we are only considering consequence between sentences). Thus every rule of the form (4) is admissible. Let G be any sentence in the language of arithmetic for which neither $\vdash_Q G$ nor $\vdash_Q \neg G$ (Gödel's Theorem guarantees the existence of such a G). Now $(\mathit{true}(G) \supset \mathit{true}(\neg\, 0 = 1)) \supset \mathit{true}(G \Rightarrow \neg\, 0 = 1)$ is clearly the translation of a rule of the form (4) and thus of an admissible rule. However, $\not\vdash_{(\Sigma_A, \mathcal{A}_Q)} (\mathit{true}(G) \supset \mathit{true}(\neg\, 0 = 1)) \supset \mathit{true}(G \Rightarrow \neg\, 0 = 1)$ (and thus $\not\vdash_{(\Sigma_A, \mathcal{A}_Q)}$ DT). This fact can be proved by considering a $(\Sigma_A, \mathcal{A}_Q)$-model, $(W_Q, =)$, defined as $(W_Q, \leq_Q)$ but with the discrete partial order. We do not go into details, but it is easy to prove that in this model that the world Mod_Q refutes the sentence. Proposition 6.11 will show that this fact can not be proved using $(W_Q, \leq_Q)$.

As there are admissible rules which are unprovable in $(\Sigma_A, \mathcal{A}_Q)$, we would like to know at least that $(\Sigma_A, \mathcal{A}_Q)$ is safe for $\mathit{Adm}_{\vdash_Q}$. Unfortunately, $(W_Q, \leq_Q)$ fails the condition of Corollary 6.7. To see this, consider the world (where $\mathbf{N}$ is the standard model of arithmetic):

$$\mathcal{S}^\dagger \quad = \quad \{\mathcal{M} \mid \mathcal{M} \text{ is model of Q not elementary equivalent to } \mathbf{N}\}$$

Lemma 6.8 *$\vdash_Q \varphi$ if and only if for all $\mathcal{M} \in \mathcal{S}^\dagger$, $\mathcal{M} \models \varphi$.*

Proof. The left-to-right implication is by classical soundness. For the converse suppose the contrary. Then there is a φ such that, for all $\mathcal{M} \in \mathcal{S}^\dagger$, $\mathcal{M} \models \varphi$, but $\not\vdash_Q \varphi$. By classical completeness there exists a model $\mathcal{M}$ of Q such that $\mathcal{M} \models \neg\varphi$. This model cannot be in $\mathcal{S}^\dagger$, so it must be elementary equivalent to $\mathbf{N}$. It is now clear that for any true sentence ψ of arithmetic, for all models $\mathcal{M}$ of Q, $\mathcal{M} \models \neg\varphi \Rightarrow \psi$. Therefore, again by completeness, $\vdash_Q \neg\varphi \Rightarrow \psi$. But then $\mathrm{Q} \cup \{\neg\varphi\}$ gives a finite axiomatisation of the true sentences of arithmetic, which is impossible. □

So $\mathit{Th}_{\vdash_Q}(\emptyset) = \mathit{Th}_{\models}(\mathcal{S}^\dagger) = \mathit{form}(\mathcal{S}^\dagger)$ (the first equality is by the lemma, the second is by Lemma 5.5). Now let T_N be the set of all true sentences of arithmetic. Clearly $T_N \supseteq \mathit{form}(\mathcal{S}^\dagger)$. However, there can be no $\mathcal{S}$ with $\mathcal{S}^\dagger \leq_Q \mathcal{S}$ such that $\mathit{form}(\mathcal{S}) = T_N$. For such an $\mathcal{S}$ must be non-empty (as $\mathit{form}(\emptyset) = \mathcal{L}_A$), but any $\mathcal{M} \in \mathcal{S}$ must satisfy the negation of some sentence in T_N, as it can not be elementary equivalent to $\mathbf{N}$. So the condition of Corollary 6.7 does indeed fail.

One way of showing that $(\Sigma_A, \mathcal{A}_Q)$ is safe for $\mathit{Adm}_{\vdash_Q}$ is to modify the definition of $(W_Q, \leq_Q)$. Redefine:

$$W_Q \quad = \quad \{\mathit{Mods}_{\models}(T) \mid T \in \mathit{Theories}_{\vdash_Q}\}$$

The rest of the model is constructed as before. This time Corollary 6.7 does apply as *form* is now an isomorphism from $(W_Q, \leq_Q)$ to $(\mathit{Theories}_{\vdash_Q}, \subseteq)$. A different and more detailed proof will be given below, exploiting properties of the implication connective.

When a logic has implication and satisfies both modus ponens and the deduction theorem, it turns out that all admissible rules are reducible to instances of these two cases. Suppose then that $(\mathcal{L}, \vdash)$ is a logic satisfying these conditions. Thus the following are both subsets of $\mathit{Adm}_{\vdash}$:

$$\begin{aligned} \mathit{MP}_S &= \{\mathit{true}(\varphi \Rightarrow \psi) \supset \mathit{true}(\varphi) \supset \mathit{true}(\psi) \mid \varphi, \psi \in \mathcal{L}\} \\ \mathit{DT}_S &= \{(\mathit{true}(\varphi) \supset \mathit{true}(\psi)) \supset \mathit{true}(\varphi \Rightarrow \psi) \mid \varphi, \psi \in \mathcal{L}\} \end{aligned}$$

Again we assume that $(\Sigma, \mathcal{A})$ is an adequate encoding of the logic.

Proposition 6.9 *If, for all* $\Phi \in MP_S \cup DT_S$, $\vdash_{(\Sigma,\mathcal{A})} \Phi$ *then* $\vdash_{(\Sigma,\mathcal{A})} \mathcal{R}^*$ *if and only if* $\mathcal{R}$ *is admissible.*

Proof. The 'only if' direction is by Proposition 6.2. For the 'if' direction we first define some notation. Given a sequent, $\mathcal{S} = (\varphi_1, \ldots, \varphi_m, \varphi)$ define:

$$\chi(\mathcal{S}) \quad = \quad \varphi_1 \Rightarrow \ldots \Rightarrow \varphi_m \Rightarrow \varphi$$

Now suppose $\mathcal{R}$, of the general form (3), is admissible. Defining $\Delta = \{\chi(\mathcal{S}_1), \ldots, \chi(\mathcal{S}_n)\}$, we have, for all $1 \leq i \leq n$, that (by m_i applications of modus ponens) $\Delta, \varphi_{i1}, \ldots, \varphi_{im_i} \vdash \psi_i$ (where $\mathcal{S}_i = (\varphi_{i1}, \ldots, \varphi_{im_i}, \psi_i)$). In short, all the $\mathcal{S}_i$ are Δ-valid. So, by the admissibility of $\mathcal{R}$, $\Delta, \varphi_1, \ldots, \varphi_m \vdash \varphi$ (where $\mathcal{S} = (\varphi_1, \ldots, \varphi_m, \varphi)$). Now, by m applications of the deduction theorem, $\Delta \vdash \varphi_1 \Rightarrow \ldots \Rightarrow \varphi_m \Rightarrow \varphi$, ie. $\chi(\mathcal{S}_1), \ldots, \chi(\mathcal{S}_n) \vdash \chi(\mathcal{S})$. Hence, by adequacy, $\mathit{true}(\chi(\mathcal{S}_1)), \ldots, \mathit{true}(\chi(\mathcal{S}_n)) \vdash_{(\Sigma,\mathcal{A})} \mathit{true}(\chi(\mathcal{S}))$. But now, using DT_S for the left-hand side and MP_S for the right, it is easy to see that $\mathcal{S}_1^*, \ldots, \mathcal{S}_n^* \vdash_{(\Sigma,\mathcal{A})} \mathcal{S}^*$. So clearly $\vdash_{(\Sigma,\mathcal{A})} \mathcal{R}^*$. □

Corollary 6.10 $(\Sigma, \mathcal{A})$ *is safe for* $Adm_{\vdash}$ *if and only if it is safe for* $MP_S \cup DT_S$.

Proof. The left-to-right implication is obvious. Suppose then that $(\Sigma, \mathcal{A})$ is safe for $MP_S \cup DT_S$. The result follows immediately from the above proposition applied to $(\Sigma, \mathcal{A} \cup MP_S \cup DT_S)$. □

We can now give the promised proof that $(\Sigma_A, \mathcal{A}_Q)$ is safe for $Adm_{\vdash_Q}$. $(\Sigma_A, \mathcal{A}_Q)$ is a logic with implication satisfying both modus ponens and the deduction theorem. As $\mathcal{A}_Q$ contains the following axiom:

$$\forall \varphi{:}o.\ \forall \psi{:}o.\ \mathit{true}(\varphi \Rightarrow \psi) \supset \mathit{true}(\varphi) \supset \mathit{true}(\psi)$$

it is clear that, for all $\Phi \in MP_S$, $\vdash_{(\Sigma_A,\mathcal{A}_Q)} \Phi$. Thus, by Corollary 6.10, it is enough to show that $(\Sigma_A, \mathcal{A}_Q)$ is safe for DT_S. In fact we will show that $(\Sigma_A, \mathcal{A}_Q \cup \{\mathrm{DT}\})$ is adequate.

Proposition 6.11 $(W_Q, \leq_Q)$ *is a* $(\Sigma_A, \mathcal{A}_Q \cup \{DT\})$*-model.*

Proof. We need only show that, for all $\mathcal{S} \in W_Q$, $\mathcal{S} \models DT$. Suppose then that we have a world $\mathcal{S}$ and an environment ρ with $\rho(\varphi)_{\mathcal{S}} = \{a_{\mathcal{M}}\}_{\mathcal{M} \in \mathcal{S}}$ and $\rho(\psi)_{\mathcal{S}} = \{b_{\mathcal{M}}\}_{\mathcal{M} \in \mathcal{S}}$ (both these in $[\![o]\!]_{\mathcal{S}}$) such that $\mathcal{S} \models_\rho \mathit{true}(\varphi) \supset \mathit{true}(\psi)$. Let $\mathcal{M}$ be any element of $\mathcal{S}$. Clearly $\mathcal{S} \leq_Q \{\mathcal{M}\}$ so if $\{\mathcal{M}\} \models_\rho \mathit{true}(\varphi)$ then $\{\mathcal{M}\} \models_\rho \mathit{true}(\psi)$. Thus $a_{\mathcal{M}} = \mathrm{ff}$ or $b_{\mathcal{M}} = \mathrm{tt}$. But then, by Lemma 5.4, $[\![\varphi \Rightarrow \psi]\!]^\rho_{\mathcal{S}} = \{\mathrm{tt}\}_{\mathcal{M} \in \mathcal{S}}$. So $\mathcal{S} \models_\rho \mathit{true}(\varphi \Rightarrow \psi)$ as required. □

We have already shown that $(W_Q, \leq_Q)$ has the property required by Proposition 5.2, so the faithfulness (and hence adequacy) of $(\Sigma_A, \mathcal{A}_Q \cup \{\mathrm{DT}\})$ is immediate.

It is easy to show that Proposition 6.11 implies, by the soundness of the meta-logic together with Proposition 6.9, that $(W_Q, \leq_Q) \models \mathcal{R}^*$ if and only if $\mathcal{R}$ is admissible. As we have shown that $(W_Q, \leq_Q)$ fails the condition of Proposition 6.6, the condition is seen to be sufficient but not necessary.

It is also clear, using Proposition 6.9 and the established adequacy result, that:

$$\vdash_{(\Sigma_A,\mathcal{A}_Q \cup \{DT\})} \mathcal{R}^* \quad \text{iff} \quad \mathcal{R} \text{ is admissible}$$

So this presentation proves all admissible rules.

The rather simple connection of Proposition 6.9, showing that the provable admissibility of any inference rule is reducible to modus ponens and the deduction theorem, is a testimony to the (over-)simplicity of our notion of inference rule. With a reasonable notion of schematic inference rule the situation is likely to be a good deal more interesting. Still, there are enough surprises even with the simpler notion (such as the potential problem with safety) to justify our investigation at this level.

The most natural way of extending this work to deal with more complex notions of rule would be to consider first more elaborate notions of logic. One possible development would be to include language structure in the definition of logic, allowing for both connectives and quantifiers. Another extension would be to consider logics with other notions of consequence (such as multiple judgements). At the logical level these notions are already established (see [5] for example). However, such ideas have yet to be approached using the semantic methods of this paper.

At least, whatever extensions to the notion of logic are considered, the semantics of section 3 is already in place to be applied. However, whether or not the fruitful connections of sections 5 and 6 will generalise remains to be seen.

Acknowledgements

I gratefully acknowledge the support of my supervisors, Gordon Plotkin and David Pym.

References

[1] A. Avron, F. Honsell, and I. Mason. Using typed lambda calculus to implement formal systems on a machine. LFCS Report Series ECS-LFCS-87-31, Laboratory for the Foundations of Computer Science, Computer Science Department, University of Edinburgh, 1987.

[2] J. Barwise. Axioms for abstract model theory. *Annals of Mathematical Logic*, 7, 1974.

[3] N. G. de Bruijn. A survey of the project automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 589 – 606. Academic Press, 1980.

[4] A. Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD theses, University of Pennsylvania, 1989.

[5] P. Gardner. *Representing Logics in Type Theory.* PhD Thesis, Department of Computer Science, University of Edinburgh, 1992.

[6] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings of 2nd Annual Symposium on Logic in Computer Science*, pages 194–204, 1987.

[7] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and the λ-Calculus.* London Mathematical Society, Student Texts 1. Cambridge University Press, 1986.

[8] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27 – 57, 1975.

[9] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume II, pages 365 – 458. Elsevier Science Publishers, 1990.

[10] J. C. Mitchell and E. Moggi. Kripke-style models for typed lambda calculus. *Journal of Pure and Applied Logic*, 51:99–124, 1991.

[11] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory.* Oxford University Press, 1990.

[12] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–396, 1989.

[13] P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49:1284–1299, 1984.

[14] D. S. Scott. On engendering an illusion of understanding. *Journal of Philosophy*, 68:787–807, 1971.

[15] A. Tarski. *Logic, Semantics, Metamathematics.* Oxford University Press, 1956.

[16] A.S. Troelstra. Metamathematical investigations of intuitionistic arithmetic and analysis. *Lecture notes in mathematics*, 344, 1973.

[17] J. van Bentham. Correspondence theory. In D. Gabbay and F. Guenther, editors, *Handbook of Philosophical Logic*, volume II, pages 167 – 47. D. Reidel Publishing Company, 1984.

A Normalization Proof for an Impredicative Type System with Large Elimination over Integers

Benjamin Werner*
INRIA – Rocquencourt

August 1992

Abstract

We prove strong normalization for a type system that includes system F, dependent types, primitive integers and the ability to define types by primitive recursion over integers. We give some applications motivating this feature.

Introduction

The aim of this note is to show how one can prove strong normalization for a type system in which types can be defined by case and by recursion over some inductive structure (here integers), thus setting the bases for a consistency proof of systems like the one implemented in Coq [4]. Let us sketch the technical motivations for these systems.

Need for inductive types

In [3], Coquand and Paulin give a uniform presentation of inductive definitions, which can be added to the Calculus of Constructions (or nearly any other GTS). To simplify, one may say that this definition scheme generalizes the primitive integers of Gödel's system T [10, 9]. It is well-known that in an impredicative framework, as is the Calculus of Constructions, inductive structures (integers, lists, sums, products) may be defined internally, using encodings like Church's integers [8, 9, 12]. However, Coquand and Paulin listed some drawbacks of this technique, thus motivating the introduction of their extension; essentially:

- Efficiency: for Church's integers, the predecessor cannot be calculated in constant time.
- The induction scheme (in the case of integers, Peano's axiom) cannot be proven internaly. Moreover the induction principle does not correspond to recursion anymore.
- It is impossible to prove $0 \neq 1$ in the system.

In what follows, we are mainly focusing on the third point. Let us try to explicit it: proving $0 \neq 1$ means precisely exhibiting a predicate P over natural numbers, such that one can prove $P(0)$ and the negation of $P(1)$. In the pure Calculus of Constructions there is no hope to define such a predicate, because ultimately, all we can do is quantify over all predicate variables

*benjamin.werner@inria.fr

$(\forall\alpha : \mathsf{Nat}\to Prop)$. The right and elegant way to formalize this argument, is to use the erasure map which transforms any type (resp. proof) of the Calculus of Constructions into a type of F_ω (resp. a term of F_ω inhabiting the corresponding type). Suppose we had a closed proof p of the proposition
$0 \neq 1 \equiv (\forall P{:}\mathsf{Nat}\to Prop.(P\ 0)\to\ (P\ \mathsf{S}(0)))\to(\forall Q{:}Prop.Q)$
then by mapping p into $F\omega$ we would get a closed term inhabiting the type:$(\forall\alpha : *.\alpha\to\alpha)\to(\forall\beta : *.\beta)$ which, we know, is impossible, as there is no closed term of type $\forall\alpha : *.\alpha$.

Large elimination schemes

So it is the mechanism for defining dependent types in CoC which appears to be unsatisfactory. Here we want to enhance it, at least in the case of inductive structures like natural numbers. Coquand and Paulin suggest to introduce a second elimination scheme for each inductive structure, which allows to build types through structural recursion over, say, a natural number. Let us illustrate this point. In system T, there exist an elimination scheme rec, with the following corresponding reduction rules:
$(\mathsf{rec}\ 0\ t_0\ t_S)\ \triangleright\ t_0$
$(\mathsf{rec}\ (\mathsf{S}\ t)\ t_0\ s_S)\ \triangleright\ (t_S\ t\ (\mathsf{rec}\ t\ t_0\ t_S))$

The natural type of rec in the Calculus of Constructions is precisely Peano's axiom:
$\forall n{:}\mathsf{Nat}.\forall P{:}\mathsf{Nat}\to Prop.(P\ 0)\to(\forall m{:}\mathsf{Nat}.(P\ m)\to(P\ (\mathsf{S}\ m)))\to(P\ n)$

Coquand and Paulin's suggestion can be understood as the introduction of a second elimination scheme Rec, with the same reduction rules, but of the following type:

$\forall n{:}\mathsf{Nat}.\forall T{:}\mathsf{Nat}\to Type.(T\ 0)\to(\forall m{:}\mathsf{Nat}.(T\ m)\to(T\ (\mathsf{S}\ m)))\to(T\ n)$

This means we can define a type (*i.e.* an object of type *Prop*) recursively over an integer. For instance, a predicate trivially verified by 0 and not verified by (S 0) could be:
$\lambda n{:}\mathsf{Nat}.(\mathsf{Rec}\ n\ \lambda n{:}\mathsf{Nat}.Prop\ \forall P{:}Prop.(P\to P)\ \lambda m{:}\mathsf{Nat}.\lambda q{:}Prop.\forall P{:}Prop.P)$

We will call Rec the *Large* elimination scheme, by opposition to the usual *small* elimination scheme rec [1].

Of course, this is generalized to all inductive types [2]. The question we try to address here, is the normalization, and hence the consistency, of type systems including such features. In what follows we restrict ourselves to the simplest possible system including such a feature combined with polymorphism.

1 The system

Therefore, the calculus defined below can be seen as the system F enriched by:

- natural numbers "à la system T"
- dependent types
- the essential novelty which is the Rec operator shown above.

In other words, it is the type system $\lambda P2$ with primitive natural numbers and the two elimination schemes. Note that dependent types are not strictly speaking necessary for the definition of the system, but without them, types defined using Rec would have no interesting inhabitants.

[1] This terminology is, to our knowledge, due to Thorsten Altenkirch.
[2] Actually all inductive types with only monomorphic constructors, also called "small" inductive types.

We should also mention that Jan Smith proposed to add large elimination over integers to Martin-Löf's Type Theory. Actually, the system he describes in [14] is quite similar to the one given below, so the idea of our normalization proof should also apply to his extension.

The syntax

In order to simplify as much as possible the normalization proof, we chose a little more complicated syntax than the usual GTS style presentations [7] (and also than the one of the few examples above). We introduce a syntactic distinction between the *term variables* and the *predicate variables*. Moreover every predicate variable comes along with its *arity* or *degree*: a variable of type $Prop$ will be of degree 0, a variable of type $\mathsf{Nat}\rightarrow Prop$ will be of degree 1, etc. Consequently the inference rules are more numerous and slightly redundant, but the whole construction of interpretating the predicates in terms on *reducibility candidates* is more straightforward.

We consider as given a set Vt of *term variables* whose elements will generally be written as x, and, for any natural number i, a set VT_i of predicate variables of degree i which will denote the symbol α_i. All these sets are supposed distinct and the equality over them decidable. the symbol α will be used to designate any element of VT which is defined as $\bigcup_{i\in\mathbb{N}} VT_i$.

We now define the set of proof-terms (noted as t), the sets of predicates of degree i (written T_i) and of kinds of degree i (written K_i):

$$
\begin{array}{rcl}
t & ::= & x \mid \lambda^{T_0}x.t \mid (t\ t) \mid \Lambda^{K_i}\alpha_i.t \mid (t\ T_i) \mid \mathsf{0} \mid \mathsf{S}\{t\} \mid \mathsf{rec}\{t,t,t\} \\
T_0 & ::= & \alpha_0 \mid \mathsf{Nat} \mid (x:T_0)T_0 \mid (\alpha_i:K_i)T_0 \mid (T_1\ t) \mid \mathsf{Rec}\{t,T_0,\alpha_0.T_1\} \\
T_{i+1} & ::= & \alpha_{i+1} \mid \lambda^{T_0}x.T_i \mid (T_{i+2}\ t) \\
K_0 & ::= & Prop \\
K_{i+1} & ::= & (x:T_0)K_i
\end{array}
$$

Furthermore we will call *objects* the elements of the union of the sets defined above. The substitution is defined for all kind of variables in the usual way and noted $t[x \setminus t']$ (or $t[\alpha \setminus T]$, etc). A *type* is a predicate of degree 0. We will not deal with alpha-conversion.

Definition We define the $\beta\iota$-reduction by:

$(\beta_{t1})\quad (\lambda^T x.t_1\ t_2) \mapsto t_1[x \setminus t_2]$

$(\beta_{t2})\quad (\Lambda^K \alpha.t\ T) \mapsto t[\alpha \setminus T]$

$(\beta_T)\quad (\lambda^{T'} x.T\ t) \mapsto T[x \setminus t]$

$(\iota_{t\mathsf{0}})\quad \mathsf{rec}\{\mathsf{0}, t_0, t_S\} \mapsto t_0$

$(\iota_{t\mathsf{S}})\quad \mathsf{rec}\{\mathsf{S}\{t\}, t_0, t_S\} \mapsto (t_S\ t\ \mathsf{rec}\{t, t_0, t_S\})$

$(\iota_{T\mathsf{0}})\quad \mathsf{Rec}\{\mathsf{0}, T_0, \alpha.T_1\} \mapsto T_0$

$(\iota_{T\mathsf{S}})\quad \mathsf{Rec}\{\mathsf{S}\{t\}, T_0, \alpha.T_1\} \mapsto (T_1[\alpha \setminus \mathsf{Rec}\{t, T_0, \alpha.T_1\}]\ t)$

We will write $M \rhd M'$ to say that M rewrites to M' by $\beta\iota$-reduction of a subterm, $M \rhd^i M'$ designing a sequence of i reductions. The transitive-reflexive closure of $\rhd$ will denote the symbol $\rhd^*$ and the transitive-symmetrical-reflexive closure will be designed by the infix symbol $=_{\beta\iota}$.

Remark The reductions preserve the syntactic class. So we can speak of the arity of an $=_{\beta\iota}$ equivalence class.

Lemma 1 *The reduction is Church-Rosser in the set of objects.*

Proof By the usual Tait–Martin-Löf method. ∎

Definition A *context* (generally written Γ) is a list of pairs composed either by a term variable and a type (x, T_0) or a predicate variable and a kind of the same degree (α_i, K_i). The empty context is written $[]$.

Type system: specific rules

$$(\textsc{Nat})\ [] \vdash \mathsf{Nat}{:}\mathit{Prop} \qquad (\textsc{zero})\ [] \vdash \mathsf{0}{:}\mathsf{Nat} \qquad (\textsc{successor})\ \frac{\Gamma \vdash t{:}\mathsf{Nat}}{\Gamma \vdash \mathsf{S}\{t\}{:}\mathsf{Nat}}$$

$$(\textsc{rec})\ \frac{\Gamma \vdash t{:}\mathsf{Nat} \qquad \Gamma \vdash T{:}\mathsf{Nat}{\to}\mathit{Prop} \qquad \Gamma \vdash t_0{:}(T\ \mathsf{0}) \qquad \Gamma \vdash t_S{:}(n{:}\mathsf{Nat})(T\ n){\to}(T\ \mathsf{S}\{n\})}{\Gamma \vdash \mathsf{rec}\{t, t_0, t_S\}{:}(T\ t)}$$

$$(\textsc{Rec})\ \frac{\Gamma \vdash t{:}\mathsf{Nat} \qquad \Gamma \vdash T_0{:}\mathit{Prop} \qquad \Gamma{::}(\alpha_0, \mathit{Prop}) \vdash T_S{:}\mathsf{Nat}{\to}\mathit{Prop}}{\Gamma \vdash \mathsf{Rec}\{t, T_0, \alpha_0.T_S\}{:}\mathit{Prop}}$$

GTS Rules (λP_2)

$$(\textsc{Prop})\ [] \vdash \mathit{Prop}{:}\mathit{Type} \qquad (\textsc{Kind})\ \frac{\Gamma :: (x, T) \vdash K : \mathit{Type}}{\Gamma \vdash (x:T)K : \mathit{Type}}$$

$$(\textsc{Prod})\ \frac{\Gamma \vdash T_0{:}\mathit{Prop} \qquad \Gamma{::}(x, T_0) \vdash T_0'{:}\mathit{Prop}}{\Gamma \vdash (x{:}T_0)T_0'{:}\mathit{Prop}} \qquad \frac{\Gamma \vdash K_i{:}\mathit{Type} \qquad \Gamma{::}(\alpha_i, K_i) \vdash T_0{:}\mathit{Prop}}{\Gamma \vdash (\alpha_i{:}K_i)T_0{:}\mathit{Prop}}$$

$$\frac{\Gamma \vdash T_0{:}\mathit{Prop} \qquad \Gamma{::}(x, T_0) \vdash K_i{:}\mathit{Type}}{\Gamma \vdash (x{:}T_0)K_i{:}\mathit{Type}}$$

$$(\textsc{Abs})\ \frac{\Gamma \vdash (x{:}T_0)T_0'{:}\mathit{Prop} \qquad \Gamma{::}(x, T_0) \vdash t{:}T_0'}{\Gamma \vdash \lambda^{T_0}x.t : (x{:}T_0)T_0'}$$

$$\frac{\Gamma \vdash (x{:}T_0)K_i{:}\mathit{Type} \qquad \Gamma{::}(x, T_0) \vdash T_i{:}K_i}{\Gamma \vdash \lambda^{T_0}x.T_i : (x{:}T_0)K_i}$$

$$\frac{\Gamma \vdash (\alpha_i{:}K_i)T_0{:}Prop \qquad \Gamma{::}(\alpha_i,K_i) \vdash t{:}T_0}{\Gamma \vdash \Lambda^{K_i}\alpha_i.t : (\alpha_i{:}K_i)T_0}$$

$$(\textsc{App})\ \frac{\Gamma \vdash t{:}(x:T_1')T_1 \qquad \Gamma \vdash t'{:}T_1'}{\Gamma \vdash (t\ t') : T_1[x \setminus t']} \qquad \frac{\Gamma \vdash t{:}(\alpha_i{:}K_i)T_0 \qquad \Gamma \vdash T_i{:}K_i}{\Gamma \vdash (t\ T_i){:}T_0[\alpha_i \setminus T_i]}$$

$$\frac{\Gamma \vdash T_{i+1}{:}(x:T_0)K_i \qquad \Gamma \vdash t{:}T_0}{\Gamma \vdash (T_{i+1}\ t){:}K_i[x \setminus t]}$$

$$(\textsc{redT})\ \frac{\Gamma \vdash T{:}Prop \qquad T =_{\beta\iota} T' \qquad \Gamma \vdash T'{:}Prop \qquad \Gamma \vdash t{:}T}{\Gamma \vdash t{:}T'}$$

$$(\textsc{redK})\ \frac{\Gamma \vdash K{:}Type \qquad K =_{\beta\iota} K' \qquad \Gamma \vdash K'{:}Type \qquad \Gamma \vdash T{:}K}{\Gamma \vdash T{:}K'}$$

$$(\textsc{weak})\ \frac{\Gamma \vdash V{:}S \qquad \Gamma \vdash u{:}U}{\Gamma{::}(a,V) \vdash u{:}U} \qquad \begin{array}{l}(a,V,S) \in (Vt \times T_0 \times \{Prop\}) \cup (VT_i \times K_i \times \{Type\}) \\ \text{and } (u,U) \in (t \times T_0) \cup (T_i \times K_i)\end{array}$$

$$(\textsc{var})\ \frac{\Gamma \vdash u{:}U \qquad (a,V) \text{ in } \Gamma}{\Gamma \vdash a{:}V} \qquad \begin{array}{l}(u,U) \in (t \times T_0) \cup (T_i \times K_i) \\ \text{and } (a,V) \in (Vt \times T_0) \cup (VT_i \times K_i)\end{array}$$

2 Pure proof-terms

We define the term algebra Λ by:
$u\ ::=\ x \mid (u\ u) \mid \lambda x.u \mid \mathsf{0} \mid \mathsf{S}\{u\} \mid \mathsf{rec}\{u,u,u\}$

We will call pure terms the elements of Λ. The symbols u, v, u' etc will be used to denote pure terms.

We define the following map from proof-terms to Λ:

$$\begin{array}{rcl}
\lceil x \rceil & \equiv & x \\
\lceil \lambda^T x.t \rceil & \equiv & \lambda x.\lceil t \rceil \\
\lceil t\ t' \rceil & \equiv & (\lceil t \rceil\ \lceil t' \rceil) \\
\lceil \Lambda^K \alpha.t \rceil & \equiv & \lceil t \rceil \\
\lceil t\ T \rceil & \equiv & \lceil t \rceil \\
\lceil \mathsf{0} \rceil & \equiv & \mathsf{0} \\
\lceil \mathsf{S}\{t\} \rceil & \equiv & \mathsf{S}\{\lceil t \rceil\} \\
\lceil \mathsf{rec}\{t,t_0,t_s\} \rceil & \equiv & \mathsf{rec}\{\lceil t \rceil, \lceil t_0 \rceil, \lceil t_S \rceil\}
\end{array}$$

We may easily map the reduction defined over proof-terms to reductions over pure terms. We will call $\mathcal{N}$ the set of strongly normalizable pure terms.

Lemma 2 *The $\beta\iota$-reduction is Church-Rosser in the set of pure terms.*

Proof Tait–Martin-Löf method again; it is the usual proof for system T. ■

The reader familiar with normalization proofs will understand that we will start by proving strong normalization for pure terms. Hence, the proof below has a lot of similarities with the ones found in [11, 7]. There is however a major difference due to the presence of the large elimination scheme: we cannot get rid of the dependent types in the proof. Actually the main originality of our proof is the way we interpret of the types defined by recursion over integers. On the other hand, we should also say that it is perfectly possible to do a similar proof for dependent types without first proving strong normalization for pure terms; for example [2] could very well be adapted for our type system. So the choice between typed or untyped reducibility candidates is still a matter of convenience and taste.

Definition Pure terms of the following forms are said to be *neutral*: x, $(u_1\ u_2)$ and $\mathsf{rec}\{u, u_0, u_S\}$.

Definition *A predicate interpretation* $\mathcal{C}$ of degree 0 is a set of pure terms. A predicate interpretation of degree $i+1$ is a total function f mapping any pure term to a predicate interpretation of degree i.

Definition An *interpretation* $\mathcal{I}$ is a pair composed of:

- A total function mapping a term to every term variable.
- A total function mapping a predicate interpretation of degree i to every predicate variables of degree i, for any i.

If $\mathcal{I} = (f, g)$ we will write $\mathcal{I}(x)$ (resp. $\mathcal{I}(\alpha)$) for $f(x)$ (resp. $g(\alpha)$). The first component of any interpretation straightforwardly defines a substitution over the free variables of any pure term. If u is a pure term, we will write $u[\mathcal{I}]$ for the substituted term.

We will write $\mathcal{I}; x \leftarrow u$ (resp. $\mathcal{I}; \alpha \leftarrow \mathcal{C}$) for the interpretation which has the same values than $\mathcal{I}$ on all points but x for which it will have value u (resp. α for which it will have value $\mathcal{C}$). Note that all variables are substituted *in parallel*: $\mathcal{I}; x \leftarrow u; y \leftarrow u' = \mathcal{I}; y \leftarrow u'; x \leftarrow u$, provided that $x \neq y$. A simple but essential property is: $u[x \setminus u'][\mathcal{I}] = u[\mathcal{I}; x \leftarrow u'[\mathcal{I}]]$. Note also that $\lceil t[\alpha\setminus T]\rceil[\mathcal{I}] = \lceil t\rceil[\mathcal{I}] = \lceil t\rceil[\mathcal{I}; \alpha \leftarrow \mathcal{C}$.

Definition *A reducibility candidate* of degree 0 is a set of pure terms such that:

- $\mathcal{C}$ is closed under $\beta\iota$-reduction.
- Every element of $\mathcal{C}$ is strongly normalizable.
- let u be a neutral pure term. If for every term u' such that $u \rhd^1 u'$ we have $u' \in\mathcal{C}$, then $u \in\mathcal{C}$. Note that a reducibility candidate of degree 0 is never empty, since it contains all the term variables.

For $n \geq 1$, a reducibility candidate of degree n is a predicate interpretation f of degree n such that for any pure term u, $f(u)$ is a reducibility candidate of degree $n-1$ such that if $u =_{\beta\iota} u'$ then $f(u) = f(u')$.

Definition Let T be any predicate of any degree i. By structural induction over T we define $|T|_{\mathcal{I}}$ for any interpretation $\mathcal{I}$. If $i = 0$ then $|T|_{\mathcal{I}}$ is a set of pure terms and a total function from terms to some set otherwise.

- If $T = \alpha_i$, then $|T|_{\mathcal{I}} \equiv \mathcal{I}(\alpha_i)$.
- If $T = \mathsf{Nat}$, then then $|T|_{\mathcal{I}} \equiv \mathcal{N}$.
- If $T = (x{:}T')T''$, then $|T|_{\mathcal{I}} \equiv \{u, \forall u' \in |T'|_{\mathcal{I}}.(u\ u') \in |T''|_{\mathcal{I};x\leftarrow u'}\}$.

- If $T = (\alpha_i : K_i)T'$, then $|T|_{\mathcal{I}} \equiv \bigcap_{\mathcal{C}_i} |T'|_{\mathcal{I};\alpha_i \leftarrow \mathcal{C}_i}$ where the intersection ranges over all reducibility candidates (*not* all interpretations) of degree i.

- If $T = (T_{i+1}\ t)$, then $|T|_{\mathcal{I}} \equiv |T_{i+1}|_{\mathcal{I}}(\lceil t \rceil[\mathcal{I}])$.

- If $T = \lambda^{T_0} x.T'$, then $|T|_{\mathcal{I}}$ is the function that to any term u associates $|T'|_{\mathcal{I};x \leftarrow u}$.

- If $T = \mathsf{Rec}\{t, T_0, \alpha_0.T_1\}$, then $|T|_{\mathcal{I}}$ is defined by cases on the normalizability [3] of $\lceil t \rceil[\mathcal{I}]$:
  - If $\lceil t \rceil[\mathcal{I}]$ admits no normal form, then $|T|_{\mathcal{I}} \equiv \mathcal{N}$.
  - If $\lceil t \rceil[\mathcal{I}]$ (weakly) normalizes [4] to u', we define the set $\mathcal{C}(v, \mathcal{I}, T_0, T_1, \alpha_0)$ by structural recursion over the (normal) pure term v:
    * $\mathcal{C}(\mathsf{0}, \mathcal{I}, T_0, T_1, \alpha_0) \equiv |T_0|_{\mathcal{I}}$.
    * $\mathcal{C}(\mathsf{S}\{v'\}, \mathcal{I}, T_0, T_1, \alpha_0) \equiv |T_1|_{\mathcal{I};\alpha_0 \leftarrow \mathcal{C}(v', \mathcal{I}, T_0, T_1, \alpha_0)}(v')$.
    * $\mathcal{C}(v, \mathcal{I}, T_0, T_1, \alpha_0) \equiv \mathcal{N}$ otherwise.

    We then may define $|T|_{\mathcal{I}} \equiv \mathcal{C}(u', \mathcal{I}, T_0, T_1, \alpha_0)$.

Lemma 3 *For any x, T, t and $\mathcal{I}$, we have $|T[x \setminus t]|_{\mathcal{I}} = |T|_{\mathcal{I};x \leftarrow \lceil t \rceil[\mathcal{I}]}$.*

PROOF By straightforward structural induction over T:

- If $T = (y{:}T')T''$ with $x \neq y$ and y not free in t, then $T[x \setminus t] = (y{:}T'[x \setminus t])T''[x \setminus t]$. The induction hypothesis applies for $|T'[x \setminus t]|_{\mathcal{I}}$ and $|T''[x \setminus t]|_{\mathcal{I};y \leftarrow u_0}$ for any u_0. The result follows immediately.

- If $T = (T_{i+1}\ t_0)$ then
$$|T[x \setminus t]|_{\mathcal{I}} = |(T_{i+1}[x \setminus t]\ t_0[x \setminus t])|_{\mathcal{I}}$$
which by induction hypothesis is equal to
$$|T_{i+1}|_{\mathcal{I};x \leftarrow \lceil t \rceil[\mathcal{I}]}(\lceil t_0 \rceil[\mathcal{I}; x \leftarrow \lceil t \rceil[\mathcal{I}]]) = |T_{i+1}|_{\mathcal{I};x \leftarrow \lceil t \rceil[\mathcal{I}]}(\lceil t_0[x \setminus t] \rceil[\mathcal{I}]).$$
The result follows.

- If $T = \lambda^{T_0} y.T'$ with $x \neq y$ and y not free in t, then $|T[x \setminus t]|_{\mathcal{I}}$ is the function that to u_0 associates $|T'[x \setminus t]|_{\mathcal{I};y \leftarrow u_0}$ which is equal to
$$|T'|_{\mathcal{I};y \leftarrow u_0; x \leftarrow \lceil t \rceil[\mathcal{I}; y \leftarrow u_0]} = |T'|_{\mathcal{I};x \leftarrow \lceil t \rceil[\mathcal{I}]; y \leftarrow u_0}.$$

- If $T = \mathsf{Rec}\{t_0, T_0, \alpha_0.T_1\}$ then $|T[x \setminus t]|_{\mathcal{I}}$ is defined by cases over the normalizability of $\lceil t_0 \rceil[x \setminus \lceil t \rceil][\mathcal{I}]$ which is equal to $\lceil t_0 \rceil[\mathcal{I}; x \leftarrow \lceil t \rceil[\mathcal{I}]]$. The result follows with straightforward use of the induction hypothesis in the different cases.

All other cases are immediate. ■

Lemma 4 *For any T, α_i, T_i and $\mathcal{I}$, we have $|T[\alpha_i \setminus T_i]|_{\mathcal{I}} = |T|_{\mathcal{I};\alpha_i \leftarrow |T_i|_{\mathcal{I}}}$.*

[3] This is obviously a classical resonning. It is possible to make this proof constructive, but this would imply to define $|T|_{\mathcal{I}}$ provided some well-chosen conditions. We believe the classical proof is simpler and more enlighting.

[4] We recall that the normal form is unique, as the reduction is Church-Rosser; note that in this proof we do need CR for the pure terms, but not for the annotated proof terms.

PROOF Again we proceed by simple structural induction over T. We only detail the Rec case; all the other ones are straightforward.

If $T = \mathsf{Rec}\{t, T_0, \alpha_0.T_1\}$, we have seen that $\lceil t\rceil[\mathcal{I}] = \lceil t[\alpha_i \setminus T_i]\rceil = \lceil t\rceil$. The case where $\lceil t\rceil$ admits no normal form is thus trivial. Now for the case where $\lceil t\rceil$ does admit a normal form, we prove by induction over the (normal) pure term u that

$$\mathcal{C}(u, \mathcal{I}, T_0[\alpha_i \setminus T_i], T_1[\alpha_i \setminus T_i], \alpha_0) = \mathcal{C}(u, \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0).$$

Again, we do not cope with α-conversion, thus admitting that α_0 is not free in T_1.

- If $u = \mathsf{0}$,
$$\mathcal{C}(\mathsf{0}, \mathcal{I}, T_0[\alpha_i \setminus T_i], T_1[\alpha_i \setminus T_i], \alpha_0) = |T_0[\alpha_i \setminus T_i]|_{\mathcal{I}}$$
which is equal to
$$|T_0|_{\mathcal{I};\alpha_i \leftarrow |T_i|_{\mathcal{I}}}$$
because of the (first) induction hypothesis and by definition equal to
$$\mathcal{C}(0, \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0)$$

- If $u = \mathsf{S}\{u'\}$, the (second) induction states that
$$\mathcal{C}(u', \mathcal{I}, T_0[\alpha_i \setminus T_i], T_1[\alpha_i \setminus T_i], \alpha_0) = \mathcal{C}(u', \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0)$$
By definition
$$\mathcal{C}(\mathsf{S}\{u'\}, \mathcal{I}, T_0[\alpha_i \setminus T_i], T_1[\alpha_i \setminus T_i], \alpha_0) = |T_1[\alpha_i \setminus T_i]|_{\mathcal{I};\alpha_0 \leftarrow \mathcal{C}(u', \mathcal{I}, T_0[\alpha_i \setminus T_i], T_1[\alpha_i \setminus T_i], \alpha_0)}$$
which, because of the equality above, is equal to
$$\begin{aligned}&|T_1[\alpha_i \setminus T_i]|_{\mathcal{I};\alpha_0 \leftarrow \mathcal{C}(u', \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0)}\\ &\qquad = |T_1|_{\mathcal{I};\alpha_0 \leftarrow \mathcal{C}(u', \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0); \alpha_i \leftarrow |T_i|_{\mathcal{I};\alpha_0 \leftarrow \mathcal{C}(u', \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0)}}\end{aligned}$$
But as α_0 is not free in T_1, this last term can be simplified into:
$$|T_1|_{\mathcal{I};\alpha_i \leftarrow |T_i|_{\mathcal{I}}; \alpha_0 \leftarrow \mathcal{C}(u', \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0)}$$
Which is precisely the definition of $\mathcal{C}(\mathsf{S}\{u'\}, \mathcal{I}; \alpha_i \leftarrow |T_i|_{\mathcal{I}}, T_0, T_1, \alpha_0)$.

- The case where u neither reduces to $\mathsf{0}$ nor to $\mathsf{S}\{u'\}$ is trivial.

■

Definition An predicate interpretation f of degree $i > 0$ is said to be *$\beta\iota$-invariant* if for any u and u' such that $u =_{\beta\iota} u'$, one has $f(u) = f(u')$. Furthermore if $i > 1$ then $f(u)$ has to be $\beta\iota$-invariant also.

An interpretation $\mathcal{I}$ is said to be $\beta\iota$-invariant if and only if for any α_{i+1}, $\mathcal{I}(\alpha_{i+1})$ is $\beta\iota$-invariant.

Lemma 5 *Let $\mathcal{I}$ be a $\beta\iota$-invariant interpretation. Then the following holds:*

- *For any T of degree at least 1, $|T|_{\mathcal{I}}$ is $\beta\iota$-invariant.*

- *Furthermore, for any T, x u and u' such that $u =_{\beta\iota} u'$, one has $|T|_{\mathcal{I};x\leftarrow u} = |T|_{\mathcal{I};x\leftarrow u'}$.*

PROOF By structural induction over T.

- $T = \alpha$ is immediate.
- $T = \mathsf{Nat}$ is immediate.
- $T = (y : T')T''$. The second induction hypothesis applies for T' and T''. Therefore

$$\begin{array}{rcl} |T|_{\mathcal{I};x\leftarrow v} & = & \{u, \forall u' \in |T'|_{\mathcal{I};x\leftarrow v}.(u\ u') \in |T''|_{\mathcal{I};x\leftarrow v}\} \\ & = & \{u, \forall u' \in |T'|_{\mathcal{I};x\leftarrow v'}.(u\ u') \in |T''|_{\mathcal{I};x\leftarrow v'}\} \\ & = & |T|_{\mathcal{I};x\leftarrow v'} \end{array} .$$

- $T = (\alpha_i : K_i)T'$. for any reducibility candidate $\mathcal{C}$of degree i, $\mathcal{I};\alpha_i \leftarrow \mathcal{C}$ is $\beta\iota$-invariant because $\mathcal{C}$is invariant if $i > 0$. Therefore, by induction hypothesis, $|T'|_{\mathcal{I};\alpha_i\leftarrow\mathcal{C}}$ is also $\beta\iota$-invariant. The result follows.
- $T = (T_{i+1}\ t)$. $|T|_{\mathcal{I};x\leftarrow u} = |T_{i+1}|_{\mathcal{I};x\leftarrow u}\ (\lceil t\rceil[\mathcal{I};x \leftarrow u])$ which is equal to $|T|_{\mathcal{I};x\leftarrow u'}$ by the induction hypothesis. Furthermore, if $i > 0$, $|T|_{\mathcal{I}}$ is $\beta\iota$-invariant, because $|T_{\subset+1}|_{\mathcal{I}}$ is.
- $T = \lambda^{T_0}x.T'$ is similar.
- $T = \mathsf{Rec}\{t, T_0, \alpha_0.T_1\}$. Because of Church-Rosser for pure terms, $\lceil t\rceil[\mathcal{I};x \leftarrow v]$ has a normal form if and only if $\lceil t\rceil[\mathcal{I};x \leftarrow v']$ has. Furthermore if these normal forms exist, they are equal. The result follows by considering the different cases of the definition of $|T|_{\mathcal{I}}$.

■

Definition An interpretation $\mathcal{I}$ is said to be a *candidate interpretation* if for any α_i, $\mathcal{I}(\alpha_i)$ is a reducibility candidate of degree i.

Lemma 6 *For any predicate T and any candidate interpretation $\mathcal{I}$, $|T|_{\mathcal{I}}$ is a reducibility candidate of the same degree than T.*

PROOF Structural induction over T again. Most of the cases reflect what is done in other proofs of the literature. Of course, we use the fact that the previous lemma applies, as $\mathcal{I}$ is $\beta\iota$-invariant.

- If $T = \alpha_i$. By definition: $\mathcal{I}$ is a candidate interpretation.
- If $T = \mathsf{Nat}$. We have to prove that $\mathcal{N}$ is indeed a reducibility candidate. It is easy to see that it is closed under $\beta\iota$-reduction and it is of course contained by itself. Also a term that only reduces to strongly normalizable terms is itself strongly normalizable.
- If $T = (x : T')T''$, then $|T'|_{\mathcal{I}}$ and $|T''|_{\mathcal{I};x\leftarrow u'}$ are reducibility candidates of degree 0 for any pure term u', by induction hypothesis. Let u be an element of $|T|_{\mathcal{I}}$ and u' any element of $|T'|_{\mathcal{I}}$. Then as $(u\ u')$ is strongly normalizable, so is u. Also if $u \triangleright u''$, then $(u\ u') \triangleright (u''\ u')$; thus $(u''\ u')$ is in $|T''|_{\mathcal{I};x\leftarrow u'}$ and hence u'' is in $|T|$.

  Finally, given any neutral term u_0 that reduces only to elements of $|T|$, we may prove by induction over the maximum number of reduction steps of u' that $(u_0\ u')$ is indeed element of $|T''|_{\mathcal{I};x\leftarrow u'}$. It is neutral and we show it only reduces to elements of $|T''|_{\mathcal{I};x\leftarrow u'}$:

- $(u_0\ u') \triangleright (u'_0\ u')$ but then u'_0 is element of $|T|_{\mathcal{I}}$ and thus $(u'_0\ u')$ is element of $|T''|_{\mathcal{I};x\leftarrow u'}$.
- $(u_0\ u') \triangleright (u_0\ u'')$ but the maximum number of reduction steps inside of u'' is strictly less than in u'. So we can apply the induction hypothesis and $|T''|_{\mathcal{I};x\leftarrow u''} = |T''|_{\mathcal{I};x;\leftarrow u'}$.

- For the case $T = (\alpha_i : K_i)T'$, we have to prove that any intersection of reducibility candidates (of degree 0) is still a reducibility candidate. This is done straightforwardly.
- The case $T = (T_{i+1}\ t)$ is a simple application of the induction hypothesis.
- If $T = \lambda^{T_0}x.T_i$, the induction hypothesis says that for any term u $|T_i|_{\mathcal{I};x\leftarrow u}$ is a reducibility candidate of degree i. The previous lemma ensures that the function $|T|_{\mathcal{I}}$ is invariant through $\beta\iota$-conversion of its argument.
- If $T = \mathsf{Rec}\{t, T_0, \alpha_0.T_1\}$:

  If $\lceil t\rceil[\mathcal{I}]$ admits no normal form, we saw that $\mathcal{N}$ was a reducibility candidate.

  If $\lceil t\rceil[\mathcal{I}]$ normalizes to u we prove by induction over u that the set $\mathcal{C}(u, \mathcal{I}, T_0, T_1, \alpha_0)$ as it is defined in the definition of $|T|_{\mathcal{I}}$ is indeed a reducibility candidate.

  - if $u = \mathsf{0}$ then $|T_0|_{\mathcal{I}}$ is a candidate by induction hypothesis.
  - if $u = \mathsf{S}\{u'\}$, then $\mathcal{C}(u', \mathcal{I}, T_0, T_1, \alpha_0)$ is a reducibility candidate by induction hypothesis. Therefore $\mathcal{I}; \alpha_0 \leftarrow \mathcal{C}(u')$ is a candidate interpretation, and $|T_1|_{\mathcal{I};\alpha_0\leftarrow\mathcal{C}(u')}$ is a reducibility candidate.
  - For all other cases we saw that $\mathcal{N}$ was a reducibility candidate.

■

Definition Consider a context Γ. A candidate interpretation $\mathcal{I}$ is said to be *admissible* for Γ if and only if for any (x, T_0) of Γ one has $\mathcal{I}(x) \in |T_0|_{\mathcal{I}}$.

Theorem 1 *Consider a context Γ and $\mathcal{I}$, an interpretation which is admissible for Γ. Then the following holds:*

- *for any well-formed judgement $\Gamma \vdash t : T_0$ we have $\lceil t\rceil[\mathcal{I}] \in |T_0|_{\mathcal{I}}$.*
- *for any well-formed judgement $\Gamma \vdash T_i : K_i$ and for any judgement $\Gamma \vdash T'_i{:}K_i$, if $T_i \triangleright T'_i$ then $|T_i|_{\mathcal{I}} = |T'_i|$.*

Proof Now for the fight. We proceed by structural induction over the derivation of the typing judgement; the different cases to consider are each of the typing rules. In what follows, and if t (resp. t', t_1, etc) is a proof term, then u (resp. u', u_1, etc) is the corresponding substituted pure term $\lceil t\rceil[\mathcal{I}]$ (resp. $\lceil t'\rceil[\mathcal{I}]$, $\lceil t_1\rceil[\mathcal{I}]$, etc).

- The rules Nat, zero, successor are trivial.
- The rec rule. $\lceil \mathsf{rec}\{t, t_0, t_S\}\rceil[\mathcal{I}]$ is neutral and we show it only reduces to elements of $|(T\ t)|_{\mathcal{I}}$. As in [9] we reason by induction over $\nu(u) + \nu(u_0) + \nu(u_S) + l(u')$ where ν is the maximal number of reduction steps in a term and $l(u')$ the length (number of symbols) of the normal form u' of u.

  $\mathsf{rec}\{u, u_0, u_S\}$ may reduce to:

– $\mathsf{rec}\{u'', u'_0, u'_S\}$ where … which is element of $|(T\ t'')|_{\mathcal{I}} = |(T\ t)|_{\mathcal{I}}$ by induction hypothesis.
– u_0 if $u = \mathsf{0}$; which is element of $|(T\ \mathsf{0})|_{\mathcal{I}}$ by induction hypothesis.
– $(u_S\ u''\ \mathsf{rec}\{u'', u_0, u_S\})$ if $u = \mathsf{S}\{u''\}$.
By induction hypothesis we know that

$$\mathsf{rec}\{u'', u_0, u_S\} \in |(T\ u'')|_{\mathcal{I}}.$$

Therefore

$$(u_S\ u'') \in |(T\ t'') \rightarrow (T\ \mathsf{S}\{t''\})|_{\mathcal{I}}$$

and this implies that

$$(u'_S\ u''\ \mathsf{rec}\{u'', u'_0, u'_S\}) \in |(T\ \mathsf{S}\{t''\})|_{\mathcal{I}} = |(T\ t)|_{\mathcal{I}}.$$

- Formation rule for $(t_1\ t_2)$. The induction hypothesis ensures that $u_2 \in |T^2|_{\mathcal{I}}$ and $u_1 \in |(x : T^2)T^1|_{\mathcal{I}}$. Hence, by definition of $|(x : T^2)T^1|_{\mathcal{I}}$ and because of the adequacy lemma, we know that $(u_1\ u_2) \in |T^1|_{\mathcal{I};x\leftarrow u_2} = |T^1[x \setminus t_2]|_{\mathcal{I}}$.

- Formation rule for $\lambda^{T_0}x.t$. The induction Hypothesis ensures that for any term $u_0 \in |T_0|_{\mathcal{I}}$ we have $u[\mathcal{I}; x \leftarrow u_0] \in |T'|_{\mathcal{I};x\leftarrow u_0}$. To prove that $\lceil \lambda^{T_0}x.t \rceil[\mathcal{I}] = \lambda x.u \in |(x : T_0)T'|_{\mathcal{I}}$, we have to show that given any term $u_0 \in |T_0|_{\mathcal{I}}$, $(\lambda x.u\ u_0) \in |T'|_{\mathcal{I};x\leftarrow u_0}$. We do so by induction over $\nu(u) + \nu(u_0)$. The term $(\lambda x.u\ u_0)$ is neutral; it may reduce to:

  – $(\lambda x.u'\ u'_0)$ where u' and u'_0 are respectively reducts of u and u_0. This term is element of $|T'|_{\mathcal{I};x\leftarrow u_0}$ because of the last induction hypothesis.
  – $u[x \setminus u_0]$. But this is element of $|T'|_{\mathcal{I};x\leftarrow u_0}$, as we have seen.

- Formation rule for $\Lambda^{K_i}\alpha_i.t$. We have to prove that $\lceil \Lambda^{K_i}\alpha_i.t \rceil[\mathcal{I}] = \lceil t \rceil[\mathcal{I}] \in |(\alpha_i : K_i)T_0|_{\mathcal{I}}$. Let $\mathcal{C}_i$ be any reducibility candidate of degree i. Then $\mathcal{I}; \alpha \leftarrow \mathcal{C}_i$ is admissible for $\Gamma :: (\alpha_i, K_i)$ and so $u \in |T_0|_{\mathcal{I};\alpha\leftarrow\mathcal{C}_i}$.

- Formation rule for $(t\ T)$. We have $\lceil t \rceil[\mathcal{I}] = u \in |(\alpha : K)T_0|_{\mathcal{I}}$, and so $\lceil t\ T \rceil[\mathcal{I}] = u \in |T_0|_{\mathcal{I};\alpha\leftarrow|T|_{\mathcal{I}}} = |T_0[\alpha \setminus T]|_{\mathcal{I}}$.

- Formation rule for $\lambda^{T_0}x.T$. $\lambda^{T_0}x.T$ may only reduce to:

  – $\lambda^{T_0}x.T'$ with $T \rhd T'$. By induction hypothesis, for any u', $|T|_{\mathcal{I};x\leftarrow u'} = |T'|_{\mathcal{I};x\leftarrow u'}$. The result follows.
  – $\lambda^{T'_0}x.T$ with $T_0 \rhd T'_0$. By definition, $\lambda^{T'_0}x.T|_{\mathcal{I}} = |\lambda^{T_0}x.T|_{\mathcal{I}}$.

- Formation rule for $(T\ t)$. If $t \rhd t'$, we know that $|(T\ t)|_{\mathcal{I}} = |(T\ t')|_{\mathcal{I}}$. If $t \rhd T'$, by induction hypothesis, $|(T\ t)|_{\mathcal{I}} = |(T'\ t)|_{\mathcal{I}}$. Finally if $T = \lambda^{T_0}x.T'$, by definition and the adequacy lemma, we have $|(T\ t)|_{\mathcal{I}} = |T'[x \setminus t]|_{\mathcal{I}}$.

- The Rec rule. $T = \mathsf{Rec}\{t, T_0, \alpha.T_1\}$ may reduce to:

  – $\mathsf{Rec}\{t', T_0, \alpha.T_1\}$, in which case we have already seen that $|T|_{\mathcal{I}}$ remains unchanged (the reduction takes place in a proof-subterm).

- $\mathsf{Rec}\{t, T_0', \alpha.T_1\}$ or $\mathsf{Rec}\{t, T_0, \alpha.T_1'\}$. In which case the induction hypothesis easily enables us to prove that $|T|_{\mathcal{I}}$ remains unchanged (by induction over the definition of $|T|_{\mathcal{I}}$).
- T_0, if $\lceil t \rceil[\mathcal{I}] = \mathsf{0}$. but then we have $|T|_{\mathcal{I}} = |T_0|_{\mathcal{I}}$.
- $(T_1[\alpha \setminus \mathsf{Rec}\{t', T_0, \alpha.T_1\}]\ t')$, if $\lceil t \rceil[\mathcal{I}] = \mathsf{S}\{t'\}$. But by induction hypothesis, we know that $\lceil t \rceil[\mathcal{I}]$ is strongly normalizable [5]; the desired equality follows.

- The VAR rule. Immediate, by definition.
- The WEAK rule. Any interpretation which is admissible for $\Gamma :: (a, V)$ is also valid for Γ. The result follows.
- The REDT rule. All we have to check is that if T and T' are well-typed and $T =_{\beta\iota} T'$ then there is a path from T to T' which only goes through well-typed predicates. This is true because subject-reduction holds and because of the Church-Rosser property.
- The REDK rule. Similar.

■

3 Strong normalization for erased terms implies strong normalization for typed terms

As in [7] we will prove the strong normalization for all the well-typed objects, using a coding. But instead of coding our types as simply typed lambda terms, we will code the well-typed kinds and predicates as proof-terms.

In what follows, o and c_o are respectively a fresh type variable, and a fresh term variable. We suppose also that to each predicate variable α we may associate a distinct *term variable* $\overline{\alpha}$, which, we will suppose, has not been used before.

Definition. We define the application ρ from kinds to predicates:

- $\rho(Prop) = o$
- $\rho((x:T)K) = (x:T)\rho(K)$

Definition For any context Γ, we define the context $\overline{\Gamma}$ by:

- $\overline{[]} = [] :: (o, Prop) :: (c_o, o)$
- $\overline{\Gamma :: (x,T)} = \overline{\Gamma} :: (x,T)$
- $\overline{\Gamma :: (\alpha,K)} = \overline{\Gamma} :: (\alpha,K) :: (\overline{\alpha}, \rho(K))$

We may also define:
$D_{Prop} \equiv c_o \quad D_{(x:T)K} \equiv \lambda^T x.D_K$
which of type $\rho(K)$ in any context in which $\rho(K)$ is well-typed.
$C_{Prop} \equiv o \quad C_{(x:T)K} \equiv \lambda^T x.C_K$
which is of type K in any context in which K is well-typed.

[5]Note that this is the only case where we use the reducibility property for a proof term inside a predicate.

Now, to any object M (proof-term, predicate or kind) we may associate the following proof-term $[M]$:

$$\begin{array}{rcl}
[x] & = & x \\
[(t\ t')] & = & ([t]\ [t']) \\
[\lambda^T x.t] & = & \lambda^T x.(\lambda^o _.[t]\ [T]) \\
[\lambda^K \alpha.t] & = & \lambda^K \alpha.(\lambda^o _.[t]\ [K]) \\
[(t\ T)] & = & (\lambda^{\rho(K)} _.[t]\ [T]\ T) \\
[\mathsf{rec}\{t, t_0, t_S\}] & = & \mathsf{rec}\{[t], [t_0], [t_S]\} \\
[\alpha] & = & \overline{\alpha} \\
[(T\ t)] & = & ([T]\ [t]) \\
[\lambda^{T'} x.T] & = & \lambda^{T'} x.(\lambda^o _.[T]\ [T']) \\
[\mathsf{Rec}\{t, T_0, \alpha.T_1\}] & = & rec\{[t], [T_0], \lambda^o \overline{\alpha}.([T_1][\alpha \setminus o])\} \\
[(x : T_1)T_2] & = & (\lambda^o _.\lambda^{T_1 \mapsto o} _.c_o\ [T_1]\ (\lambda^{T_1} x.[T_2])) \\
[(\alpha : K)T] & = & (\lambda^{\rho(K)} \overline{\alpha}.\lambda^K \alpha.\lambda^o _.[T]\ [K]\ D_K\ C_K) \\
[Prop] & = & o \\
[(x : T)K] & = & (\lambda^o _.[K]\ [T])
\end{array}$$

.

Lemma 7 *If* $\Gamma \vdash t{:}T$ *then* $\overline{\Gamma} \vdash [t]{:}T$.

Proof By structural induction over the typing judgement. For induction loading, one also proves that if $\Gamma \vdash T{:}K$ then $\overline{\Gamma} \vdash [T]{:}\rho(K)$ and $\overline{\Gamma} \vdash [K]{:}o$. ■

Lemma 8 *If* $t \rhd t'$, *then* $\lceil [t] \rceil \rhd^+ \lceil [t'] \rceil$.

Proof By induction over t, and because of the facts:

- $\lceil [t[x \setminus t']] \rceil = \lceil [t] \rceil [x \setminus \lceil [t'] \rceil]$
- $\lceil [t[\alpha \setminus T]] \rceil = \lceil [t] \rceil [\overline{\alpha} \setminus \lceil [T] \rceil]$

■

Theorem 2 *Any well-typed object is strongly normalizable.*

Proof The lemma above and theorem 1 imply that any well-typed proof-term is strongly normalizable. It is easy to show that this fact also implies the strong normalization of any well-typed kind or predicate. ■

4 Some remarks about large elimination schemes

Let us try to give some examples of useful or amusing ways to use large elimination schemes. In what follows we do not restrict large elimination schemes to integers. We will also use the notation of the system Coq[6]. Not all the examples given below can be run in the currently distributed version of the system because no large elimination is provided for computational inductive types at the moment.

[6] `[x:T]t` for abstraction and `(x:T)T'` for quantification.

4.1 Inversion of inductive predicates

In the hitherto developed Coq examples, this is probably the most widely used application of large elimination schemes. Actually it is a generalization of the $0 \neq 1$ proof. We will simply give an example, taken from the proof of sorting algorithms [4]. The problem is to define what it means for a list to be sorted. Lists are defined as usual:

```
Variable  A:Set.
Inductive Set list = nil : list | cons : A -> list -> list.
```

Given a binary predicate variable over `A`, one defines the sorting property as an *inductive predicate*:

```
Variable   inf : A -> A -> Prop.

Inductive Definition list_Lowert [a:A] : list -> Prop =
      nil_low : (list_Lowert a nil)
    | cons_low : (b:A)(l:list)(inf a b)->(list_Lowert a (cons b l)).

Inductive Definition sort : list -> Prop =
      nil_sort : (sort nil)
    | cons_sort : (a:A)(l:list)(sort l)->(list_Lowert a l)->(sort (cons a l)).
```

This definition is quite natural. The problem is that, in practice, one often needs properties like:

`(sort (cons a l))->(list_lowert a l)`

or similarly:

`(list_lowert a (cons b l))->(inf a b)`.

The point is, that this is based on the fact that `nil_low` cannot be used to prove `(sort a l)`; *i.e.* it implicitly uses the fact that `nil` and `cons` are different. Therefore, in order to prove the properties above, we define two new predicates, using large elimination, which are respectively equivalent de `list_lowert` and `sort`:

```
Definition list_Lowert2 =
   [a:A][l:list](<Prop> Match l with
                    (* nil *) True
               (* cons b l *) [b:A][l:list][H:Prop](inf a b)).

Definition sort_inv
[l:list](<Prop>Match l with
     (* nil *) True
(* cons a l *) [a:A][l:list][H:Prop](sort l)/\(list_Lowert a l)).
```

It is easy to prove the equivalences, which allows us to get the desired properties:
`(sort (cons a l))` is equivalent to `(sort_inv (cons a l))` which reduces to
`(sort l)/\(list_lowert a l)`. Similar examples can be found in [5, 13].

4.2 ML dynamics

The original motivation for large elimination schemes was to enhance the system from a logical point of view, *i.e.* being able to prove new properties. As long as the normalization problem was open, not very much attention was paid to the possibility to use them in the *computational part*. This example and the ones below are meant to show that large elimination schemes allow to type new interesting programs.

It is easy to define an inductive structure which is trivially isomorphic to the types of simple typed λ-calculus. For example:

```
Inductive Definition MLtype : Set =
   MLnat : MLtype
 | MLarrow : MLtype -> MLtype -> MLtype.
```

allows us to represent the types of functions and functionals over natural numbers. Adding other base types, sums, products, etc is of course no problem.

The point is that the large elimination scheme allows us to build the actual simple type out of its representation of type `MLtype`:

```
Definition MLtrad = [rT:MLtype]
               (<Set>Match rT with
                  (* MLnat *) nat
                  (* (MLarrow T1 T2) *)
                        [T1:MLtype][S1:Set][T2:MLtype][S2:Set](S1->S2))
    : MLtype -> Set.
```

The remarkable point is that, say, `(MLtrad (MLarrow MLnat MLnat))` will actually reduce (*i.e.* is *syntactically equivalent*) to the type `nat->nat`. This is exactly what we need in order to define *Dynamics* like in ML [1], but without any extension of the type system. A dynamic is a pair composed of a type representation and an element of the corresponding type:

```
Inductive Definition MLdyn : Set =
Dyn_intro : (T:MLtype)(MLtrad T)->MLdyn.
```

Note that this construction can be extended in order to represent, for instance, polymorphic (system F) types and dynamics.

4.3 Polymorphism

One may notice that in order to define dynamics in the paragraph above, we did not make use of polymorphism; hence this construction can be done in a first-order system. On the other hand, the new ability to represent types as first-order objects and then translate them back into types, allows us to define certain polymorphic functions. The point being, that we can quantify over type representations:

```
Definition Church = (a:MLtype)
                        (MLtrad a)
                             ->((MLtrad a)->(MLtrad a))
                                ->(MLtrad a).
Definition Church0 = [a:MLtype][x:(MLtrad a)][f:(MLtrad a)->(MLtrad a)]x.
Definition ChurchS = [n:Church]
                         [a:MLtype]
                              [x:(MLtrad a)][f:(MLtrad a)->(MLtrad a)]
                                (f (n a x f)).
```

It is worth noting that this type system:

- is strictly more expressive that ML polymorphism: we can impose that the argument of some function is polymorphic. For instance `ChurchS` can only be applied to a polymorphic argument.
- does not embed system F: we cannot represent universally quantified type like $\forall\alpha.T$ and hence we cannot apply its representation to some Church integer for instance.

4.4 Mutually recursive types

The large elimination feature offers a possible solution to the problem of mutually recursive types. Again, the idea is best explained through an example. The following definition does not, at first, fit into the framework of [3]:

```
     expr = Num : nat -> expr
          | closure : expr -> Env -> expr
and env = nil : Env
        | cons : term -> Env -> Env.
```

We may however define an inductive family of Sets indexed by booleans:

```
Inductive Definition P : bool -> Set =
  Num : nat -> (P true)
| closure : (P true)->(P false)->(P true)
| nil : (P false)
| cons : (P true)->(P false)->(P false).

Env := (P false)     expr := (P true)
```

The point is that we may now get the expected recursive elimination schemes thanks to large elimination. For example we can prove:

```
(Q:Env->Prop)(Q nil)->
   ((E:Env)(Q E)->(t:expr)(Q (cons t E)))->
           (F:Env)(Q F).
```

Acknowledgements

A lot of the presented material is due to, or has been shown to me by Christine Paulin-Mohring. I would also like to thank Hugo Herbelin and Thorsten Altenkirch for their insightful remarks.

References

[1] M. Abadi, L. Cardelli, B. Pierce, G. Plotkin. "Dynamic Typing in a Statically Typed Language." Conference version in Proceedings of POPL 1989. Extended version in ACM Transactions on Programming Languages and Systems, vol. 13, pp. 237-268, 1991.

[2] Th. Coquand, J. Gallier. "A Proof of Strong Normalization For the Theory of Constructions Using a Kripke-like Interpretation". Proceedings of the first BRA workshop on Logical Frameworks, G. Huet and G. Plotkin Eds. Antibes, 1990.

[3] Th. Coquand, Ch. Paulin-Mohring. "Inductively Defined Types". Proceedings of COLOG-88, P. Martin-Löf and G. Mints Eds. LNCS 417. Tallin, 1988.

[4] G. Dowek, A. Felty, H. Herbelin, G. Huet, Ch. Paulin-Mohring, B. Werner. "The Coq Proof Assistant User's Guide, Version 5.6." INRIA, Technical Report 134, 1991.

[5] G. Huet. "The Gildbreath Trick: A case study in axiomatisation and proof development in the Coq proof assistant." INRIA research report 1511, 1991.

[6] J. Gallier. "On Girard's Candidats de Réductibilité, *Logic and Computer Science*, P. Odifreddi (Ed.), Academic Press, pp. 123-203, London, 1990.

[7] H. Geuvers, M.-J. Nederhof. "A Modular Proof of Strong Normalization for the Calculus of Constructions". Journal of Functional Programming, 1991.

[8] J.-Y. Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". Thèse d'Etat, Université Paris VII, 1972.

[9] J.-Y. Girard. "Proofs and Types". Cambridge University Press, 1989.

[10] K. Gödel. "Uber eine bisher noch nicht benützte Erweiterung des finiten Standpunktes." Reprinted with English translation in the Collected Works, vol. 2, pp. 240-251, Oxford University Press, 1990. Original paper in Dialectica, Vol. 12, 1958.

[11] J.-L. Krivine. "Lambda-calcul, types et modèles". Masson, 1990.

[12] Ch. Paulin-Mohring. "Extraction de Programmes dans le Calcul des Constructions." Thèse de Doctorat, Université Paris VII, 1989.

[13] Ch. Paulin-Mohring, B. Werner. "Synthesis of ML programs in the system Coq." Submitted to the Journal of Symbolic Computation.

[14] J. Smith. "Propositional Functions and Families of Types." Notre Dame Journal of Formal Logic, Vol. 30, number 3, 1989.