INFORMAL PROCEEDINGS OF THE 1993 WORKSHOP ON TYPES FOR PROOFS AND PROGRAMS

TYPES

Nijmegen

May 1993

Ed. Herman Geuvers

Contents

Foreword

This document is the preliminary proceedings of the workshop of the Esprit Basic Research Project 6453 "Types for Proofs and Programs" held at the University of Nijmegen, the Netherlands, from May 24th until May 28th 1993. The workshop was organised by Henk Barendregt and Herman Geuvers.

Local arrangements were made by Mariëlle van der Zandt, Erik Barendsen, Herman Geuvers and Mark Ruys.

These proceedings have been collected from LaTeX sources, using e-mail. It contains 22 papers from the 35 talks that were presented at the workshop. Very useful support in solving the LaTeX puzzles was provided by Erik Barendsen.

This document can be obtained by anonymous ftp from the University of Nijmegen: Type

`ftp ftp.cs.kun.nl`
`anonymous` (as login)
[your e-mail address] (as password)
`cd /pub/csi/CompMath/Types`
`bin`
`get NijmegenTypes.ps.Z`
`bye`

Workshop Programme

Types for Proofs and Programs, Nijmegen 1993

MONDAY

8.30	Registration (and coffee)
9.15	Welcome and Opening
9.30	Hayashi (Invited Lecture) - Representing logic in ATTT
10.15	BREAK
11.00	Coquand, Dybjer (Gothenburg) - Mechanized Model Construction and Normalization Proofs
11.45	Altenkirch (Edinburgh) - Proving Syntactic Properties of the Calculus of Constructions by Modifying Realisability Models
12.30	LUNCH
14.00	Luo (Edinburgh) - Logical Truths in Type Theory
14.30	Goguen (Edinburgh) - Metatheory for a Type Theory with Inductive types
15.00	BREAK
15.30	Ciszek (Edinburgh) - A LEGO proof of a variant of Langrange's theorem
16.00	Aczel, Barthe (Manchester) - Development of algebra in LEGO
16.30	DRINKS

TUESDAY

9.30	Boyer (Invited Lecture) - Some recent work in verification in Austin
10.15	BREAK
10.45	Helmink, Vaandrager (Eindhoven) - Verifying and Proof-Checking a communication protocol
11.30	Smith, Coquand (Gothenburg) - What is the status of pattern matching in Type Theory?
12.15	Pollack (Edinburgh) - Named and nameless variables in the Constructive Engine
12.30	LUNCH
14.00	Demos of Pollack and Jones (Edinburgh), Nipkow (Munich) and Pfenning (Pittsburgh),
15.30	BREAK
16.00	Demos of Bertot (Sophia-Antipolis), Wolfram (Oxford) and Magnusson and von Sydow (on ALF)

WEDNESDAY

9.30	Raffalli (Edinburgh) - Type System for Abstract Machine
10.00	Loader (Oxford) - Graph and PER models of lambda calculi
10.30	BREAK
11.00	Dowek, Huet, Werner (Paris) - On the definition of eta-long form in the calculi of the cube
11.30	Nipkow (Munich) - Axiomatic Type Classes
12.00	Bertot (Sophia Antipolis) - Proof by Pointing
12.30	LUNCH
14.00	Gardner (Edinburgh) - The characterization of call-by-need reduction
14.30	Pfenning (Pittsburgh) - Intersection Types for Logical Frameworks
15.00	BREAK
15.30	Dowek, Boyer (Paris, Austin) - Towards checking proof-checkers
16.00	PANEL DISCUSSION
19.30	WORKSHOP DINNER

THURSDAY

9.30	Curien (Invited Lecture) - Formal Parametric Polymorphism
10.00	Magnusson (Gothenburg) - Backtracking in ALF
10.30	BREAK
11.00	Murthy (Paris) - CPS demystified
11.45	Berardi (Turin) - A symmetric lambda calculus for “classical” program extraction
12.30	LUNCH
14.00	Ranta (Finland) - Constructive Categorial Grammar and its Implementation
14.30	Ritter (Cambridge) - Normalization for lambda calculi with explicit definitions
15.00	BREAK
15.30	Paulin-Mohring, Parent (Lyon) - Developing certified programs in the system Coq
16.00	Fridlender (Gothenburg) - Formalization of Higman’s lemma in ALF

FRIDAY

9.30	Bezem, Groote (Utrecht)
	- A formal verification of the alternating bit protocol in the Calculus of Constructions
10.00	Maharaj (Edinburgh)
	- Z specifications in Type Theory
10.30	BREAK
11.00	Tasistro (Gothenburg)
	- Extension of Martin-Löf's Theory of Types with labelled record types and subtyping
11.45	McKinna (Edinburgh)
	- Formalising Pure Type Systems
12.30	LUNCH
14.00	Miculan (Udine)
	- Natural deduction style systems for operational and axiomatic semantics
14.30	Frost (Cambridge)
	- A case-study of co-induction in Isabelle HOL
15.00	BREAK
15.10	Hofmann (Edinburgh)
	- Extensionality and quotient types in type theory
16.00	Geuvers (Nijmegen)
	- Conservativity between typed lambda calculi

List of Participants

Aczel	Peter	petera@cs.man.ac.uk
Altenkirch	Thorsten	alti@dcs.ed.ac.uk
Anderson	Penny	anderson@capa.inria.fr
Barendregt	Henk	henk@cs.kun.nl
Barendsen	Erik	erikb@cs.kun.nl
Barthe	Gilles	giles@computer-science.manchester.ac.uk
Berardi	Stefano	barba@di.unito.it
Bertot	Yves	bertot@paprika.inria.fr
Betarte	Gustavo	gustun@cs.chalmers.se
Bezem	Marc	bezem@phil.ruu.nl
Boyer	Robert	boyer@cs.utexas.edu
Bruijn, de	N.G.	debruijn@win.tue.nl
Burstall	Rod	rb@dcs.ed.ac.uk
Ciszek	Petr	
Coquand	Thierry	coquand@cs.chalmers.se
Curien	Pierre Louis	curien@dmi.ens.fr
Despeyroux	Joelle	jd@bagheera.inria.fr
Dowek	Gilles	dowek@pomerol.inria.fr
Dybjer	Peter	peterd@cs.chalmers.se
Fridlender	Daniel	frito@cs.chalmers.se
Frost	Jacob	Jacob.Frost@cl.cam.ac.uk
Gardner	Philippa	pag@dcs.ed.ac.uk
Gaspes	Veronica	vero@cs.chalmers.se
Geuvers	Herman	herman@cs.kun.nl
Goguen	Healfdene	hhg@dcs.ed.ac.uk
Groote	Jan Friso	jfg@phil.ruu.nl
Hayashi	Susumu	hayashi@whale.math.ryukoku.ac.jp
Helmink	Leen	helmink@prl.philips.nl
Hemerik	Kees	hemerik@win.tue.nl
Herbelin	Hugo	herbelin@margaux.inria.fr
Hofmann	Martin	mnhofman@immd7.informatik.uni-erlangen.de
Huet	Gerard	huet@margaux.inria.fr
Jones	Claire	ccmj@dcs.ed.ac.uk
Kahn	Gilles	kahn@zephir.inria.fr
Loader	Ralph C.	loader@maths.ox.ac.uk
Luo	Zhaohui	zl@dcs.ed.ac.uk
Magnusson	Lena	lena@cs.chalmers.se
Maharaj	Savi	svm@dcs.ed.ac.uk
Manoury	Pascal	manoury@margaux.inria.fr
McKinna	James	jhm@dcs.ed.ac.uk
Meche	Patrick	Patrick.Meche@cl.cam.ac.uk
Miculan	Marino	miculan@di.unipi.it
Murthy	Chet	murthy@margaux.inria.fr

Nipkow	Tobias	nipkow@informatik.tu-muenchen.de
Nordstrom	Bengt	bengt@cs.chalmers.se
Parent	Catherine	Catherine.Parent@lip.ens-lyon.fr
Parigot	Michel	parigot@logique.jussieu.fr
Paulin-Mohring	Christine	cpaulin@FRENSL61.bitnet
Peratto	Patricia	
Pfenning	Frank	Frank_ Pfenning@ALONZO.TIP.CS.CMU.EDU
Poll	Erik	erik@win.tue.nl
Pollack	Randy	rap@dcs.ed.ac.uk
Raffalli	Christophe	cr@dcs.ed.ac.uk
Ranta	Aarne	fiaara@uta.fi
Richardson	Alexis	Alexander.Richardson@prg.oxford.ac.uk
Ritter	Eike	Eike.Ritter@cl.cam.ac.uk
Ruys	Marc	markr@cs.kun.nl
Sellink	Alex	sellink@phil.ruu.nl
Simonot	Marianne	simonot@logique.jussieu.fr
Slind	Konrad	slind@informatik.tu-muenchen.de
Smith	Jan	smith@cs.chalmers.se
Streicher	Thomas	streich@informatik.uni-muenchen.de
Sydow, von	Bjorn	sydow@cs.chalmers.se
Szasz	Nora	nora@cs.chalmers.se
Tasistro	Alvaro	tato@cs.chalmers.se
Vaandrager	Frits	frits@cwi.nl
Werner	Benjamin	benjamin.werner@inria.fr
Wolfram	David	David.Wolfram@prg.oxford.ac.uk

Proving Strong Normalization of CC by Modifying Realizability Semantics *(Extended Abstract)*

Thorsten Altenkirch
Department of Computer Sciences
Chalmers University of Technology
412 96 Gothenburg, Sweden
email: alti@cs.chalmers.se

14 September 1993

1 Introduction

We will outline a strong normalization argument for the Calculus of Constructions (CC) which is obtained by modifying a Realizability interpretation (the D-set or ω-set model). By doing so we pursue two goals:

- We want to illustrate how *semantics can be used to prove properties of syntax.*
- We present a simple and extensible SN proof for CC. An example of such an extension is a system with inductive types and *large eliminations.*

This presentation is a condensed version of a part of the author's PhD thesis [Alt93a], a preliminary version has been presented in [Alt93b]. We will omit most of the proofs here, if not noted otherwise they can be found in [Alt93a].

The proof that every term typable in the calculus of constructions is strongly normalizing is known to be notoriously difficult. The original proof in Coquand's PhD thesis [2] contained a bug which was fixed in [CG90] by using a Kripke-style interpretation of contexts. Although this solves the original problem the proof remains quite intricate due to the use of typed terms and contexts. Another construction is due to Geuvers and Nederhof (see [8], p. 168), who define a forgetful, reduction-preserving map from CC to F^ω. Thereby, they reduce the problem to strong normalization for F^ω, which can be shown using the usual Girad-Tait method. The main problem with this construction is that it is not all clear, how this argument can be extended to a system with large eliminations (e.g. see [Wer92]), this is a system which allows the definition of a dependent type by primitive recursion. As an example consider the recursive definition of a type $T : \mathrm{Nat} \to \mathrm{Set}$:

$$\begin{aligned} T0 &= A \\ T(n+1) &= Tn \to Tn \end{aligned}$$

where A : Set is arbitrary. The problem is to find a non-dependent type which approximates T. The obvious choice seems to be a recursive type which solves the equation $A = A \to A$ but such a calculus would not be strongly normalizing.

Our construction avoids the use of Kripke-structures and can be understood as a generalization of the concept of saturated sets to dependent types. We only show strong normalization for Curry terms obtained by stripping the type information, but the general strong normalization result allowing reductions in type-annotations can be recovered by a simple syntactic argument. Moreover our construction also works for inductive types with large eliminations and allows to interpret types like T.

The rest of the paper is organized as follows: We review the judgement presentation of CC; then we develop a semantic framework to present interpretations of CC, which we call CC-structures. A particular example of CC-structures are saturated Λ-sets and from the soundness of this interpretation together with an additional lemma we directly derive strong normalization for Curry terms. Finally we sketch how decidability of equality can be derived. We will also discuss some problems with the η-rule in this presentation.

2 The judgement presentation of CC

CC is usually presented in the equality-as-conversion style [CH88, Bar92], i.e. the equality is just the untyped β-conversion between preterms. When we are interested in a semantical analysis of the system it seems easier to use the equality-as-judgement presentation, as it is usual for Martin-Löf Type Theory. The reason is that it is not clear how untyped conversion can be interpreted semantically. Not surprisingly this presentation is used in [Str91] who studies the categorical semantics of CC.

We will also follow [Str91] in that we use a very explicit notation: we differentiate between operations on Set (usually called Prop) and types; we annotate applications and λ-abstractions with types and in one place we go even further and also annotate the codomain of a λ-abstraction. Essentially our terms are a linear notation for derivations where the applications of the conversion rule are omitted. The more implicit notation can be justified (e.g. see [Str91, Alt93a]), but semantically it seems to be more appropriate to consider the explicit presentation as the fundamental one.

We introduce precontexts Γ, pretypes σ and preterms M [1] by the following grammar.

$$\begin{array}{rcl} \Gamma & ::= & \bullet \mid \Gamma.\sigma \\ \sigma & ::= & \Pi\sigma.\sigma \mid \mathrm{Set} \mid \mathrm{El}(M) \\ M & ::= & \omega \mid \lambda\sigma(M)^{\sigma} \mid \mathrm{app}^{\sigma.\sigma}(M,M) \mid \forall\sigma.M \end{array}$$

We use de-Bruijn-indices (ω) to represent variables but in the presentation of the syntax we may use explicit names with the obvious translation.

Using names as a shorthand we denote substitution by $M[x := N], \sigma[x := M]$. We also have an operation of weakening M^{+x}, σ^{+x} which corresponds to the usual side condition that x may

[1]Note that we use the names of the syntactic classes and the alphabetically following letters also as generic names for metavariables.

not appear free in M. For a precise definition of these operations using de-Bruijn-indices see [Alt93a].

We define the following judgements: $\vdash \Gamma$ (context validity), $\Gamma \vdash \sigma$ (type validity), $\Gamma \vdash M : \sigma$ (typing), $\Gamma \vdash \sigma \simeq \tau$ (type equality) and $\Gamma \vdash M \simeq N : \sigma$ (equality). The derivable judgements are given as the least relations closed under the following rules — note that we have omitted the obvious congruence rules to save space.

$$\vdash \bullet \tag{Empty}$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \sigma}{\vdash \Gamma.x : \sigma} \tag{Compr}$$

$$\frac{\Gamma.x : \sigma \vdash \tau}{\Gamma \vdash \Pi x : \sigma.\tau} \tag{Pi}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathrm{Set}} \tag{Set}$$

$$\frac{\Gamma \vdash A : \mathrm{Set}}{\Gamma \vdash \mathrm{El}(A)} \tag{El}$$

$$\frac{\Gamma.x : \sigma \vdash A : \mathrm{Set}}{\Gamma \vdash \mathrm{El}(\forall x : \sigma.A) \simeq \Pi x : \sigma.\mathrm{El}(A)} \tag{All-Elim}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash \sigma \simeq \tau}{\Gamma \vdash M : \tau} \tag{conv}$$

$$\frac{\vdash \Gamma \qquad i \leq |\Gamma|}{\Gamma \vdash i : \Gamma(i)} \tag{var}$$

$$\frac{\Gamma.x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma(M)^{\tau} : \Pi x : \sigma.\tau} \tag{lam}$$

$$\frac{\Gamma \vdash M : \Pi\sigma.\tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{app}^{\sigma.\tau}(M, N) : \tau[N]} \tag{app}$$

$$\frac{\Gamma.x : \sigma \vdash A : \mathrm{Set}}{\Gamma \vdash \forall x : \sigma.A : \mathrm{Set}} \tag{all}$$

$$\frac{\Gamma.x : \sigma \vdash M : \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{app}^{x:\sigma.\tau}(\lambda x : \sigma(M)^{\tau}, N) \simeq M[x := N] : \tau[x := N]} \tag{beta-eq}$$

We can easily establish a number of rather trivial properties of this presentation such that all judgements are consistent with weakening and substitution - see [Alt93a] for details.

3 CC-structures and sound interpretations

We will now define a class of semantic structures which provide sound interpretations of CC. This can be seen as an set-theoretic — i.e. element-based — alternative to the categorical model constructions as presented in [Str91, Jac91] and is clearly influenced by them. We do this in two steps: we first define LF-structures (which correspond to a logical framework) and then CC-structures based on LF-structures.

Let's start with some preliminary definitions:

Definition 1 *Assuming some encoding of pairing (x, y) and projections π_1, π_2 we have the usual set-theoretic counterparts of the basic type-theoretic operations (assume A is a set and B_a a family of sets indexed by A):*

$$\begin{aligned} \Sigma a \in A.B_a &= \{(a,b) \mid a \in A, b \in B_a\} \\ \Pi a \in A.B_a &= \{f \subseteq \Sigma a \in A.B_a \mid \forall_{a \in A} \exists!_{b \in B_a} (a,b) \in f\} \end{aligned}$$

We consider application $f(x)$ as a partial operation which is defined if there is an $(x, y) \in f$ and then $f(x) = y$.

We also introduce the following notations for dependent composition and currying:

composition *Assume $f \in \Pi a \in A.\Pi b \in B_a.C_{(a,b)}$ and $g \in \Pi a \in A.B_a$ then*

$$\begin{aligned} f[g] &= a \in A \mapsto f(a)(g(a)) \\ &\in \Pi a \in A.C_{(a,g(a))} \end{aligned}$$

currying *Assume $f \in \Pi p \in (\Sigma a \in A.B_a).C_p$:*

$$\begin{aligned} \lambda(f) &= a \in A \mapsto b \in B_a \mapsto f(a,b) \\ &\in \Pi a \in A.\Pi b \in B_a.C_{(a,b)} \end{aligned}$$

Definition 2 *A universe $\mathfrak{U}$ is a class (of sets) together with an operation which assigns to every $X \in \mathfrak{U}$ a set $\overline{X}$ which we call* the extension *of X.*

We are now ready to define LF-structures, which can be used to assign meaning to a logical framework, i.e. a basic Type Theory which contains only Π-types.

Definition 3 (LF-structure)

$$\mathcal{L} = (\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}, \mathbf{1}, \Sigma, \mathrm{Sect}, \Pi)$$

with

- $\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}$ *are universes.*
- $\mathbf{1} \in \mathfrak{U}_{\mathrm{Co}}$.
- $\Sigma(X \in \mathfrak{U}_{\mathrm{Co}}, \{Y_x \in \mathfrak{U}_{\mathrm{Ty}}\}_{x \in \overline{X}}) \in \mathfrak{U}_{\mathrm{Co}}$ *s.t.* $\overline{\Sigma(X, \{Y_x\}_x)} = \Sigma x \in \overline{X}.\overline{Y_x}$

- $\mathrm{Sect}(X \in \mathfrak{U}_{\mathrm{Co}}, \{Y_x \in \mathfrak{U}_{\mathrm{Ty}}\}_{x \in \overline{X}}) \subseteq \Pi x \in \overline{X}.\overline{Y_x}$.
- $\Pi(X \in \mathfrak{U}_{\mathrm{Ty}}, \{Y_x \in \mathfrak{U}_{\mathrm{Ty}}\}_{x \in \overline{X}}) \in \mathfrak{U}_{\mathrm{Ty}}$ *s.t.* $\overline{\Pi(X, \{Y_x\}_x)} \subseteq \Pi x \in \overline{X}.\overline{Y_x}$

is an LF-structure if the following conditions are satisfied:

Unit $\overline{\mathbf{1}}$ *is a one-element set.*

Var-0 $p \in \Sigma p \in (\Sigma x \in X.Y_x).Y_{\pi 1(p)} \mapsto \pi_2(p) \in \mathrm{Sect}(\Sigma(X, \{Y_x\}_x), \{Y_{\pi 1(p)}\}_p)$

Var-succ $\dfrac{f \in \mathrm{Sect}(X, \{Z_x\}_x)}{p \in (\Sigma a \in X.Y_x) \mapsto f(\pi_1(p)) \in \mathrm{Sect}(\Sigma(X, \{Y_x\}_x), \{Z_{\pi 1(p)}\}_p)}$

app $\dfrac{f \in \mathrm{Sect}(X, \{\Pi(Y_x, \{Z_{(x,y)}\}_{y \in Y_x})\}_x) \qquad g \in \mathrm{Sect}(X, \{Y_x\}_x)}{f[g] \in \mathrm{Sect}(X, Z_{(x,g(x))})}$

lambda $\dfrac{f \in \mathrm{Sect}(\Sigma(X, \{Y_x\}_x), \{Z_p\}_p)}{\lambda(f) \in \mathrm{Sect}(X, \{\Pi(Y_x, \{Z_{(x,y)}\}_{y \in Y_x})\}_x)}$

To interpret CC we need some additional structure to interpret Set, El and in particular (ALL-ELIM); this is reflected by the following definition:

Definition 4 (CC-structures)

$$\mathcal{C} = (\mathcal{L}, \mathfrak{M}, \mathrm{SET}, \mathrm{EL}, \mathrm{EL}^{-1}, \vartheta)$$

with

- $\mathcal{L} = (\mathfrak{U}_{\mathrm{Co}}, \mathfrak{U}_{\mathrm{Ty}}, \mathbf{1}, \Sigma, \mathrm{Sect}, \Pi)$ *is an LF-structure.*
- $\mathfrak{M} \subseteq \mathfrak{U}_{\mathrm{Ty}}$ *is the subclass of* modest sets.
- $\mathrm{SET} \in \mathfrak{U}_{\mathrm{Ty}}$.
- $\mathrm{EL}(A \in \overline{\mathrm{SET}}) \in \mathfrak{M}$ *and* $\mathrm{EL}^{-1}(A \in \mathfrak{M}) \in \overline{\mathrm{SET}}$.
- $\vartheta_{X \in \mathfrak{M}} \in \overline{X} \to \overline{\mathrm{EL}(\mathrm{EL}^{-1}(X))}$.

is a CC-structure if the following conditions are satisfied:

1. $\mathrm{Sect}(X, \mathrm{SET}) = \overline{X} \to \overline{\mathrm{SET}}$.
2. $\mathrm{EL}^{-1}(\mathrm{EL}(A)) = A$.
3. $\Pi(X, \{Y_x \in \mathfrak{M}\}_x \in \mathfrak{M}) \in \mathfrak{M}$.
4. ϑ_X *is a bijection and*

$$\dfrac{f \in \mathrm{Sect}(X, \{Y_x \in \mathfrak{M}\})}{\vartheta \circ f \in \mathrm{Sect}(X, \{\mathrm{EL}(\mathrm{EL}^{-1}(Y_x))\})}$$

Given a CC-structure $\mathcal{C}$ we can define a partial interpretation:

Definition 5 (Interpretation in CC-Structures)
Let:

$$
\begin{aligned}
\Theta(X \in \mathfrak{U}) &= \begin{cases} \mathrm{EL}(\mathrm{EL}^{-1}(X) & \textit{if } X \in \mathfrak{M} \\ X & \textit{otherwise} \end{cases} \\
\tilde{\vartheta}_{X \in \mathfrak{U}}(x \in \overline{X}) &= \begin{cases} \vartheta_X(x) & \textit{if } X \in \mathfrak{M} \\ x & \textit{otherwise} \end{cases} \\
&\in \overline{\Theta(X)}
\end{aligned}
$$

We define partial interpretation functions $[\![\vdash \Gamma]\!] \underset{\sim}{\in} \mathfrak{U}_{\mathrm{Co}}$, $\{[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}\gamma \underset{\sim}{\in} \mathfrak{U}_{\mathrm{Ty}}\}_{\gamma \underset{\sim}{\in} \overline{[\![\vdash\Gamma]\!]^{\mathcal{C}}}}$ *and* $\{[\![\Gamma \vdash M]\!]^{\mathcal{C}}\gamma\}_{\gamma \underset{\sim}{\in} \overline{[\![\vdash\Gamma]\!]^{\mathcal{C}}}}$ *by induction over the structure of the syntax:* [2]

$$
\begin{aligned}
[\![\vdash \bullet]\!]^{\mathcal{C}} &:\cong \mathbf{1} \\
[\![\vdash \Gamma.\sigma]\!]^{\mathcal{C}} &:\cong \Sigma([\![\vdash \Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}) \\
[\![\Gamma \vdash \Pi\sigma.\tau]\!]^{\mathcal{C}} &:\cong \{\Theta(\Pi([\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}\gamma, \{[\![\Gamma.\sigma \vdash \tau]\!]^{\mathcal{C}}(\gamma, x)\}_x))\}_{\gamma \in [\![\vdash\Gamma]\!]^{\mathcal{C}}} \\
[\![\Gamma \vdash \mathrm{Set}]\!]^{\mathcal{C}} &:\cong \{\mathrm{SET}\}_{\gamma \in [\![\vdash\Gamma]\!]^{\mathcal{C}}} \\
[\![\Gamma \vdash \mathrm{El}(A)]\!]^{\mathcal{C}} &:\cong \{\mathrm{EL}([\![\Gamma \vdash A]\!]^{\mathcal{C}}\gamma)\}_{\gamma \in [\![\vdash\Gamma]\!]^{\mathcal{C}}} \\
[\![\Gamma \vdash i]\!]^{\mathcal{C}} &:\cong \gamma \in [\![\vdash \Gamma]\!]^{\mathcal{C}} \mapsto \pi_2(\pi_1^i(\gamma)) \\
[\![\Gamma \vdash \lambda\sigma(M)^{\tau}]\!]^{\mathcal{C}} &:\cong \tilde{\vartheta}_{[\![\Gamma\vdash\Pi\sigma.\tau]\!]^{\mathcal{C}}\gamma} \circ \lambda([\![\Gamma.\sigma \vdash M]\!]^{\mathcal{C}}) \\
[\![\Gamma \vdash \mathrm{app}^{\sigma.\tau}(M, N)]\!]^{\mathcal{C}} &:\cong (\tilde{\vartheta}^{-1}_{[\![\Gamma\vdash\Pi\sigma.\tau]\!]^{\mathcal{C}}\gamma} \circ [\![\Gamma \vdash M]\!]^{\mathcal{C}})[[\![\Gamma \vdash N]\!]^{\mathcal{C}}] \\
[\![\Gamma \vdash \forall^{\sigma}.A]\!]^{\mathcal{C}} &:\cong \{\mathrm{EL}^{-1}(\Pi([\![\Gamma \vdash \sigma]\!]\gamma, \{\mathrm{EL}([\![\Gamma.\sigma \vdash A]\!]^{\mathcal{C}}(\gamma, x))\}_x))\}_{\gamma}
\end{aligned}
$$

Note that we use ϑ and Θ to coerce the interpretation of Sets to tehir canonical meanings. This technique is already used in [Str91] to give an interpretation of CC up-to-equality instead of merely up-to-isomorphism (which would impose a coherence problem).

We have the following core theorem about CC-structures:

Theorem 1 (Soundness) $[\![\vdash \Gamma]\!]^{\mathcal{C}}$, $[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}$ *and* $[\![\Gamma \vdash M]\!]^{\mathcal{C}}$ *defines a sound interpretation of the calculus. I.e.:*

$$\frac{\vdash \Gamma}{[\![\vdash \Gamma]\!]^{\mathcal{C}} \in \mathfrak{U}_{\mathrm{Co}}}$$

$$\frac{\Gamma \vdash \sigma \qquad \gamma \in \overline{[\![\vdash \Gamma]\!]^{\mathcal{C}}}}{[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}}\gamma \in \mathfrak{U}_{\mathrm{Ty}}}$$

$$\frac{\Gamma \vdash M : \sigma}{[\![\Gamma \vdash M]\!]^{\mathcal{C}} \in \mathrm{Sect}([\![\vdash \Gamma]\!]^{\mathcal{C}}, [\![\Gamma \vdash \sigma]\!]^{\mathcal{C}})}$$

$$\frac{\Gamma \vdash \sigma \simeq \tau}{[\![\Gamma \vdash \sigma]\!]^{\mathcal{C}} = [\![\Gamma \vdash \tau]\!]^{\mathcal{C}}}$$

[2] Read $:\cong$ as: the left hand side is defined if the right hand side is defined.

$$\frac{\Gamma \vdash M \simeq N : \sigma}{[\![\Gamma \vdash M]\!]^{\mathcal{C}} = [\![\Gamma \vdash N]\!]^{\mathcal{C}}}$$

As a simple example for a CC-structure we consider the proof-irrelevance semantics of CC:

Theorem 2

1. *Let $\mathfrak{S}$ by the universe of sets with $\overline{X} = X$ and:*
   - $\mathbf{1}_{\mathcal{S}} = \{\epsilon\}$.
   - $\Sigma_{\mathcal{S}}(A, \{B_a\}_a) = \Sigma a \in A.B_a$.
   - $\Pi_{\mathcal{S}}(A, \{B_a\}_a) = \Pi a \in A.B_a$.

   $\mathcal{S} = (\mathfrak{S}, \mathfrak{S}, \mathbf{1}_{\mathcal{S}}, \Sigma_{\mathcal{S}}, \overline{\Pi}_{\mathcal{S}}, \Pi_{\mathcal{S}})$ *is an LF-structure.*

2. *Let $\mathfrak{S}GL$ denote the class of sets with at most one element and:*

   $$\mathrm{EL}_{\mathcal{S}}^{-1}(X) = \begin{cases} \emptyset & \textit{if } X = \emptyset \\ \{\emptyset\} & \textit{otherwise} \end{cases}$$

   and $\vartheta_{\mathcal{S}X}(x) = \emptyset \in \{\emptyset\}$ (Note that $\vartheta_{\mathcal{S}}$ will be never applied if X is empty.).

   $$\mathcal{S}^{+} = (\mathcal{S}, \mathfrak{S}GL, \{\emptyset, \{\emptyset\}\}, X \mapsto X, \mathrm{EL}_{\mathcal{S}}^{-1}, \vartheta_{\mathcal{S}})$$

   is a CC-structure.

Note that this simple minded model already gives us *logical consistency* simply by observing that $[\![\vdash \forall x : \mathrm{Set}.\mathrm{El}(X)]\!]^{\mathcal{S}^{+}} = \emptyset$.

4 Saturated Λ-sets and strong normalization

For any *Partial Combinatory Algebra* (PCA) D we can define the D-set model of CC. This has been studied in great detail in [Str91]. Streicher uses a categorical notion of model but it is straightforward to define an appropriate CC-structure ([Alt93a], section 3.4.).

[HO92] describe a general categorical SN proof which is based on the idea to define a PCA from the strongly normalizing terms of λ-calculus, actually they find that they have to use a weaker concept *conditionally partial combinatory algebra.*

Our approach differs in that we do not try to define a PCA at all but we present another CC-structure — saturated Λ-sets — which is motivated by D-sets and by the conventional strong normalization arguments for non-dependent calculi using *saturated sets.* It should be also noted that our development is completely elementary whereas [HO92] use a number of abstract topos-theoretic results.

Definition 6 *We denote the set of untyped λ-terms by Λ, $\rhd \subseteq \Lambda \times \Lambda$ is the usual one-step β-reduction and $\mathrm{SN} \subseteq \Lambda$ is the set of strongly normalizing (w.r.t. $\rhd$) λ-terms. $\rhd_{\mathrm{whd}} \subseteq \rhd$ is weak head-reduction, i.e. only a left-most redex not inside a λ-abstraction is reduced.* $\mathrm{Void} \subseteq \mathrm{SN}$ *is the set of strongly normalizing weak-head normal forms which are not λ-abstractions. This set can be inductively defined as:*

1. $i \in \mathrm{Void}$.

2. $$\frac{M \in \mathrm{Void} \qquad N \in \mathrm{SN}}{MN \in \mathrm{Void}}$$

3. $$\frac{M \in \mathrm{SN}}{\forall M \in \mathrm{Void}}$$

We note the following essential properties of SN:

Lemma 1 (Properties of SN**)**

1. $$\frac{M, N, M[N] \in \mathrm{SN}}{(\lambda M)N \in \mathrm{SN}}$$

2. $$\frac{M' \ \triangleright_{\mathrm{whd}} M \qquad MN \in \mathrm{SN}}{M'N \in \mathrm{SN}}$$

We will now define Λ-sets completely analogous to ω-sets or D-sets.

Definition 7 (Λ-sets)
A Λ-set X is a pair $(\overline{X}, \Vdash_X)$ with X is a set and $\Vdash_X \subseteq \Lambda \times \overline{X}$ s.t. $\forall_{x \in \overline{X}} \exists_{i \in \Lambda} i \Vdash_X x$. Λ-sets together with $\overline{X}$ constitute the universe $\mathfrak{L}AM$. $\mathfrak{L}AM^*$ is defined analogous by relpacing Λ by Λ^*, i.e. sequences of λ-terms.

We call X modest, iff

$$\forall_{x,y \in \overline{X}} \forall_{i \in \Lambda} i \Vdash_X x \wedge i \Vdash_X y \rightarrow x = y,$$

Definition 8 *Assume $G \in \mathfrak{L}AM^*$, $\{Y_\gamma \in \mathfrak{L}AM\}_{\gamma \in \overline{G}}$, $X \in \mathfrak{L}AM$, $\{Z_x \in \mathfrak{L}AM\}_{x \in \overline{X}}$ and let:*

$$\begin{array}{rcl}
\mathbf{1}_\Lambda & = & (\{\epsilon\}, \Lambda \times \{\epsilon\}) \\
& \in & \mathfrak{L}AM^* \\
\Sigma_\Lambda(G, \{Y_\gamma\}_{\gamma \in \overline{G}}) & = & (\Sigma_{\gamma \in \overline{G}} \overline{Y_\gamma}, \{(\vec{M}N, (\gamma, y)) \mid \vec{M} \Vdash_G \gamma \wedge N \Vdash_{Y_\gamma} y\}) \\
& \in & \overline{\lambda} \\
\mathrm{Sect}_\Lambda(G, \{Y_\gamma\}_{\gamma \in \overline{G}}) & = & \{f \in \Pi_{\gamma \in \overline{G}} \overline{Y_\gamma} \mid \exists_{M \in \Lambda} M \Vdash_{\mathrm{Sect}_\Lambda(G, \{Y_\gamma\}_\gamma)} f\} \\
\multicolumn{3}{c}{\textit{where } \Vdash_{\mathrm{Sect}_\Lambda(G, \{Y_\gamma\}_\gamma)} = \{(M, f) \mid \forall_{\gamma \in \overline{G}} \forall_{\vec{N} \in \overline{\lambda}} \vec{N} \Vdash_G \gamma \rightarrow M[\vec{N}] \Vdash_{Y_\gamma} f(\gamma)\}} \\
\Pi_\Lambda(X, \{Z_x\}_{x \in \overline{X}}) & = & (\{f \in \Pi_{x \in \overline{X}} \overline{Z_x} \mid \exists_{M \in \Lambda} M \Vdash_{\Pi_\Lambda(X, \{Z_x\}_x)} f\}, \Vdash_{\Pi_\Lambda(X, \{Z_x\}_x)}) \\
& \in & \mathfrak{L}AM \\
\multicolumn{3}{c}{\textit{where } \Vdash_{\Pi_\Lambda(X, \{Z_x\}_x)} = \{(M, f) \mid \forall_{x \in \overline{X}} \forall_{N \in \Lambda} M \Vdash_X x \rightarrow MN \Vdash_{Z_x} f(x)\}}
\end{array}$$

If we quotient Λ by β-conversion then the previous construction gives directly rise to a CC-structure, which is consequence of the fact that $\Lambda / \approx_\beta$ is a PCA. Here we will refrain from doing so and instead identify a substructure of Λ-sets:

Definition 9 *We call a Λ-set X saturated — $X \in \mathfrak{S}AT$ — iff the following conditions hold:*

SAT1 *Every realizer is strongly normalizing.*

$$\forall_{M \Vdash_X x} M \in \mathrm{SN}$$

SAT2 *There is a* $\bot_X \in \overline{X}$ *which is realized by every void term.*

SAT3 *The set of realizers for a certain element* x *is closed under weak head expansion inside* SN*:*

$$\forall_{M \Vdash_X x} \forall_{M' \in \mathrm{SN}} (M' \rhd_{\mathrm{whd}} M) \to (M' \Vdash_X x)$$

This can be extended to $\mathfrak{L}AM^*$*-sets by the following inductive definition:*

1. $\mathbf{1}_\Lambda \in \mathfrak{S}AT^*$.
2. $$\frac{G \in \mathfrak{S}AT^* \qquad \{X_\gamma \in \mathfrak{S}AT\}_{\gamma \in \overline{G}}}{\Sigma_\Lambda(G, \{X_\gamma\}_{\gamma \in \overline{G}}) \in \mathfrak{S}AT^*}$$

Note that for any saturated Λ-set $(\overline{X}, \Vdash_X)$ the set of realizers $\{M \mid \exists_{x \in \overline{X} M \Vdash_X x}\}$ is saturated in the conventional sense.

$\mathbf{1}_\Lambda$ and Σ_Λ are operations on saturated Λ-sets by definition but it remains to show that this is also true for Π_Λ:

Lemma 2 *Assume* $X \in \mathfrak{S}AT$, $\{Y_x \in \mathfrak{S}AT\}_{x \in \overline{X}}$ *then* $\Pi_\Lambda(X, \{Y_x\}_x) \in \mathfrak{S}AT$.

Proof:

SAT1 Assume $M \Vdash_{\Pi_\Lambda(X, \{Y_x\}_x)} f$, certainly $0 \Vdash_X \bot_X$ (**SAT2** for X). Now we know that $M0 \Vdash_{Y_{\bot_X}} f(\bot_X)$, therefore $M0 \in \mathrm{SN}$ (**SAT1** for Y_x), which implies $M \in \mathrm{SN}$.

SAT2 Assume $M \in \mathrm{Void}$, now for every $N \Vdash_X x$ we have that $MN \in \mathrm{Void}$ (**SAT1** for X and definition of Void) and therefore $MN \Vdash_{Y_x} \bot_{Y_x}$. This implies $M \Vdash_{\Pi_\Lambda(X, \{Y_x\}_x)} x \mapsto \bot_{Y_x}$, so we just set $\bot_{\Pi_\Lambda(X, \{Y_x\}_x)} = x \mapsto \bot_{Y_x}$.

SAT3 Assume $M \Vdash_{\Pi_\Lambda(X, \{Y_x\}_x)} f$, $M' \in \mathrm{SN}$ and $M' \rhd_{\mathrm{whd}} M$. For any $N \Vdash_X x$ we have that $MN \Vdash_{Y_x} f(x)$. By (App-l) $M'N \rhd_{\mathrm{whd}} MN$ and by lemma 1 (3.) $M'N \in \mathrm{SN}$. Using **SAT3** for Y_x we have that $M'N \Vdash_{Y_x} f(x)$. Therefore we have established that $M' \Vdash_{\Pi_\Lambda(X, \{Y_x\}_x)} f$.

∎

The essential idea of saturated Λ-sets is that we can prove closure under the λ-introduction rule. This is precisely the point where we need the weak β-equality in the usual D-set semantics.

Lemma 3 *Let* $G \in \mathfrak{S}AT^*, \{X_\gamma \in \mathfrak{S}AT\}_{\gamma \in \overline{G}}, \{Z_\delta \in \mathfrak{S}AT\}_{\delta \in \overline{\Sigma_\Lambda(G, \{X_\gamma\}_\gamma)}}$ *then*

$$\frac{M \Vdash_{\mathrm{Sect}_\Lambda(\Sigma_\Lambda(G, \{X_\gamma\}_\gamma), \{Z_\delta\}_\delta)} f}{\lambda M \Vdash_{\mathrm{Sect}_\Lambda(G, \{\Pi_\Lambda(X_\gamma, \{Z_{(\gamma,x)}\}_x)\}_\gamma)} \lambda(f)}$$

Proof: For any $\gamma \in \overline{G}$ and $\vec{N} \Vdash_G \gamma$ we have that

$$M[\vec{N}]^1 = M[\vec{N}0] \Vdash_{Z_{\gamma,\bot}} f(\gamma, \bot)$$

(using SAT2) and therefore $M[\vec{N}]^1 \in \mathrm{SN}$ (by SAT1).

Furthermore assume $x \in X_\gamma$ and $N' \Vdash_{X_\gamma} x$. We have that

$$M[\vec{N}]^1[N'] = M[\vec{N}N'] \Vdash_{Z_{(\gamma,x)}} f(\gamma, x)$$

Knowing $M[\vec{N}]^1, N', M[\vec{N}]^1[N'] \in \mathrm{SN}$ we can apply lemma 1 (2.) to conclude that $(\lambda M[\vec{N}]^1)N' \in \mathrm{SN}$. We can now apply SAT3 because $(\lambda M)[\vec{N}]N' = \lambda M[\vec{N}]^1)N' \rhd_{\mathrm{whd}} M[\vec{N}]^1[N']$ to see that

$$(\lambda M)[\vec{N}]N \Vdash_{Z_{(\gamma,x)}} f(\gamma, x) = \lambda(f)(\gamma)(x)$$

Therefore (by discharging the assumptions) we have that

$$\lambda M \Vdash_{\mathrm{Sect}_\Lambda(G,\{\Pi_\Lambda(X_\gamma,\{Z_{(\gamma,x)}\}_x)\}_\gamma)} \lambda(f).$$

∎

The following theorem summarizes that saturated Λ-sets constitute a CC-structure *and* that every typing derivation $\Gamma \vdash M : \sigma$ is realized by its underlying Curry-term $|M|$:

Theorem 3

1. *$\mathcal{SAT} = (\mathfrak{S}AT^*, \mathfrak{S}AT, \mathbf{1}_\Lambda, \Sigma_\Lambda, \mathrm{Sect}_\Lambda, \Pi_\Lambda)$ is an LF-structure.*

2. *Let $\mathfrak{M}_\Lambda$ be the universe of saturated modest Λ-sets, assume $R \in \mathrm{PER}(\Lambda)$ and let:*

$$\begin{array}{rcl}
\mathrm{EL}_\Lambda(R) & = & (\Lambda/R, \in) \\
\mathrm{EL}_\Lambda^{-1}(X) & = & \{(M,N) \mid \exists_{x\in\overline{X}} M \Vdash_X x \ \wedge N \Vdash_X x\} \\
\vartheta_{\Lambda X}(x \in \overline{X}) & = & \{M \mid M \Vdash_X x\} \\
\overline{\mathrm{SET}_\Lambda} & = & \{R \in \mathrm{PER}(\Lambda) \mid \mathrm{EL}(R) \in \mathfrak{S}AT\} \\
\mathrm{SET}_\Lambda & = & (\overline{\mathrm{SET}_\Lambda}, \mathrm{SN} \times \overline{\mathrm{SET}_\Lambda})
\end{array}$$

   we have that

$$\mathcal{SAT}^+ = (\mathcal{SAT}, \mathfrak{M}_\Lambda, \mathrm{SET}_\Lambda, \mathrm{EL}_\Lambda, \mathrm{EL}_\Lambda^{-1}, \vartheta_\Lambda)$$

   is a CC-structure.

3. *For any $\Gamma \vdash M : \sigma$ we have that*

$$|M| \Vdash_{\mathrm{Sect}_\Lambda([\![\Gamma]\!]^{\mathcal{SAT}^+}, [\![\Gamma\vdash\sigma]\!]^{\mathcal{SAT}^+})} [\![\Gamma \vdash M]\!]^{\mathcal{SAT}^+}$$

We need the previous two lemmas only to verify that $\mathcal{SAT}$ is an LF-structure fulfilling the self-realizing property (3.). The extension to CC-structure proceeds analogous to the development for D-sets.

We can now pick the fruits:

Corollary 1 (Strong normalization)
If $\Gamma \vdash M : \sigma$ then $|M| \in \mathrm{SN}$.

Proof: By **SAT2** we have

$$0, 1, \ldots |\Gamma| \Vdash_{[\![\Gamma]\!]^{\mathfrak{S}AT^+}} \bot, \bot \ldots \bot = \vec{\bot}$$

and by Theorem 3 (3.) we can conclude:

$$|M| = |M|[0, 1, \ldots |\Gamma|] \Vdash_{[\![\Gamma \vdash \sigma]\!]^{\mathfrak{S}AT^+} \vec{\bot}} [\![\Gamma \vdash M]\!]^{\mathfrak{S}AT^+} \vec{\bot}$$

and therefore $M \in \mathrm{SN}$ by **SAT1**. ∎

5 Decidability

We have only shown SN for Curry terms, it is not immediately obvious that this implies SN for a system with Church terms and reductions in type annotations. [3] We are also interested to establish decidability of the equality judgement with the view towards type checking.

Our approach to solve the first problem is to define a type-preserving map which blows up terms such that every reduction in a Church term can be mirrored by a reduction in a Curry term:

Definition 10 *Let*

$$\begin{array}{rcl} \bot & = & \forall x : \mathrm{Set}.x \\ M(\sigma, N) & = & \mathrm{app}^{x:\mathrm{Set}.\sigma^+}(\lambda x : \mathrm{Set}(M^{+x})^{\sigma^{+x}}, N) \end{array}$$

We now define blow $\in (M \uplus \sigma) \to (M \uplus \sigma)$*:*

$$\begin{array}{rcl} \mathrm{blow}(\Pi\sigma.\tau) & = & \mathrm{blow}(\sigma)(\mathrm{Set}, \mathrm{blow}(\tau)) \\ \mathrm{blow}(\mathrm{Set}) & = & \bot \\ \mathrm{blow}(\mathrm{El}(A)) & = & \mathrm{blow}(A) \\ \mathrm{blow}(i) & = & i \\ \mathrm{blow}(\mathrm{app}^{\sigma.\tau}(M, N)) & = & \mathrm{app}^{\sigma.\tau}(\mathrm{blow}(M), \mathrm{blow}(N))(\tau[N], \mathrm{blow}(\sigma))(\tau[N], \mathrm{blow}(\tau)) \\ \mathrm{blow}(\lambda\sigma(M)^\tau) & = & \lambda\sigma(\mathrm{blow}(M))^\tau(\Pi\sigma.\tau, \mathrm{blow}(\sigma))(\Pi\sigma.\tau, \mathrm{blow}(\tau)) \\ \mathrm{blow}(\forall\sigma.A) & = & \forall\sigma.\mathrm{blow}(A)(\mathrm{Set}, \mathrm{blow}(\sigma)) \end{array}$$

We have the following properties:

Lemma 4

1. $\dfrac{\Gamma \vdash \sigma}{\Gamma \vdash \mathrm{blow}(\sigma) : \mathrm{Set}}$

2. $\dfrac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{blow}(M) : \sigma}$

3. If $C \rhd D$ *then* $|\mathrm{blow}(C)| \rhd^+ |\mathrm{blow}(D)|$.

[3] We will denote reduction on Church-terms and -types by $\rhd$ as well.

From this it should be obvious how to derive the following (using Corollary 1):

Corollary 2

1. $$\frac{\Gamma \vdash \sigma}{\sigma \in \mathrm{SN}}$$

2. $$\frac{\Gamma \vdash M : \sigma}{M \in \mathrm{SN}}$$

In the conversion presentation the previous result would suffice to establish decidability because conversion is just defined as the transitive symmetric closure of $\rhd$. In our presentation the reasoning is a bit more intricate, because we would have to establish a *subject reduction property*, which is a non-trivial property of the system.

To avoid this we define another notion of reduction — tight reduction:

$$\mathrm{app}^{\sigma.\tau}(\lambda\sigma(M)^{\tau}, N) \ \rhd_{\mathrm{t}} M[N] \qquad \text{(BETA-RED)}$$

For $\rhd_{\mathrm{t}}$ the subject reduction property can be easily established. We can also show the weak Church Rosser property and it is easy to see that $\rhd_{\mathrm{t}}$ is strongly normalizing for derivable terms and types because $\rhd_{\mathrm{t}} \subseteq \rhd$.

Alas, this approach does not solve the problem with the η-rule. Although the strong normalization result can be easily extended to η-reduction and we also have weak Church Rosser for tight η reduction, we cannot establish subject reduction even for tight η-reduction. This result seems to depend on the *strengthening property* — i.e. that we can omit variables which do not occur in the term and type from the context — and it is not clear how to establish this property without showing subject reduction first.

6 Discussion

The essential problem in extending our strong normalization argument to a system with inductive types which allows the definition of Sets by recursion is to extend the Realizability interpretation. This corresponds to showing that initial T-algebras exist in D-set for functors given by reasonable signatures and that they are modest. Although at least the first part of this proposition seems to be folklore we could not find a satisfying presentation. In [Alt93a] we show how the D-set and the saturated Λ-set semantics can be extended to a non-algebraic inductive type with large eliminations. We claim that the same argument works for a general class of inductive definitions.

References

[Alt93a] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, 1993. forthcoming.

[Alt93b] Thorsten Altenkirch. Yet another strong normalization proof for the Calculus of Constructions. In *Proceedings of the Vintermötet*, 1993. To appear as Chalmers Techreport.

[Bar92] H.P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. 2*, pages 118 – 310. Oxford University Press, 1992.

[CG90] Thierry Coquand and Jean Gallier. A proof of strong normalization for the theory of constructions using a Kripke-like interpretation. Informal Proceedings of the First Annual Workshop on Logical Frameworks, Antibes, 1990.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76:95 – 120, 1988.

[Coq85] Th. Coquand. *Une théorie des constructions.* PhD thesis, Université Paris VII, 1985.

[Geu93] Herman Geuvers. *Logics and Type Systems.* PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[HO92] J.M.E. Hyland and C.-H. L. Ong. Modified realizability semantics and strong normalization proofs. Extended Abstract, 1992. To appear in the proceedings of the TLCA 92.

[Jac91] Bart Jacobs. *Categorical Type Theory.* PhD thesis, Katholieke Universiteit Nijmegen, 1991.

[Str91] Thomas Streicher. *Semantics of Type Theory.* Birkhäuser, 1991.

[Wer92] Benjamin Werner. A normalization proof for an impredicative type system with large eliminations over integers. In *BRA Workshop on Logical Frameworks*, 1992. Preliminary Proceedings.

CC^+: An extension of the Calculus of Constructions with fixpoints [§]

Philippe Audebaud
Ecole Normale Supérieure de Lyon
LIP-IMAG, URA CNRS 1398
46 Allée d'Italie, 69364 Lyon cedex 07, France
e-mail : paudebau@lip.ens-lyon.fr

July, 1993

Abstract

We follow an original idea suggested by Constable and Smith [6, 7] providing a way for reasoning about non terminating computations in a typed framework. A former study has been worked out within NuPrl by Smith [21]. We investigate how these ideas can be developed within the Calculus of Constructions (CC). The adaptation provides an conservative extension, denoted CC^+. Strong normalisation for β-reductions is preserved. We recover the alternate "recursive" coding for integers introduced in AF2 by Parigot [12, 13]. Thus, the computational behaviour for terms coding integers is improved. Moreover, as expected, all partial recursive functions are now definable. Relationships with primitive coding through "Church" integers within the pure Calculus is studied, giving some insights into logical expressiveness issue. 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 typed systems. The correspondence between logic and functional programming within a typed framework forces both proofs and programs to be identified with strongly normalizing lambda-terms. Even if we may content ourselves with such a strong result from a pure logical point of view, it seems clear enough that typed systems miss the point if 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.

So, connections between logic and programming need to be refined, maybe even beyond the Curry-Howard correspondence. Griffin's paper [10] emphasized the interactions between continuations (in Scheme) and classical reasoning. This starting point led to a convincing typed system, recently introduced by Parigot [14], which showed how Curry-Howard correspondence could be extrapolated to capture more programming constructs. Thus, we are suggested to approach the relations between reasoning and programming in a different way.

Same kind of connections have to be found with regard to such programming tools as unbounded loops, recursive definitions. Since we intent to study these features from a logical

[§]Invited talk at the 1992 Workshop on Types for Proofs and Programs. Båstad, June 1992

point of view, it is a natural approach to place ourselves in a typed framework. Previous studies have been presented in [11] where Mendler gave a strongly normalizing extension of polymorphic second order typed calculus, in [9] by Crole and Pitts, based on Moggi's categorical monads. However, our approach is quite different, since the kind of extension we are about to work with, requires the consideration of lambda-terms no longer normalisable. Therefore, it is much like to be compared with Constable and Mendler [8] or Parigot [13].

The starting point is the thesis that recursive structures can be coded by fixpoints. So, this paper is concerned with the study of an extension of a typed framework with a fixpoint constructor, allowing us to reason with non terminating programs or other recursive program schemes. Our study, restricted to the Calculus of Constructions (CC), actually develops in a similar way as Parigot's study of recusive integers in the AF2 framework. However, it has been initiated with Constable and Smith papers [6, 7] and Paulin-Mohring's thesis [17]. From the former, we got an original way to introduce smoothly non terminating computations in a typed framework. The last one showed how connections with primitive codings for data structures are to be understood.

This paper develops as follows. The first part gives a short overview of different possible solutions for the introduction of partial (i.e. possibly non terminating) terms and we describe how to make the last of these solution fit with the Calculus of Constructions. Then, the extension CC^+ is presented with its main metamathematical properties. The second part is devoted to the study of different internal codings for integers in our framework. Computational and logical expressiveness are emphasized throughout this presentation. A last section will be concerned with a synthesis and suggestions for further improvements.

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 typed system ; we at least assume the $\rightarrow$ constructor is available in $\mathcal{T}$. The main property we should expect from a typed system is that there exists at least one type inhabited by no term. This property insures its consistency, as a logical framework: there exists at least a proposition which is not provable.

2.1 A first attempt

Let P be any type and 0 an empty type. Given a term $f : P \rightarrow P$ and a fixpoint constructor, say $fix()$, let us allow the formation for 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 the term $\lambda x.x$ of type $P \rightarrow P$; hence $fix(\lambda x.x) : P$. Hence, 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). Martin-Löf observed that the base type $\mathbf{N}$ is now provided with a fixpoint for the successor function. But then, axiom $\forall n \quad n \neq n+1$ is no longer valid.

2.3 A general solution

In [6, 7], Constable and Smith suggested a simple and elegant solution which avoids the difficulties encountered. Moreover, it seems to fit with most of the typed systems.

New type constructor Starting from a typed system $\mathcal{T}$, we build an extension $\mathcal{T}^+$ where the same rules are allowed at the level of types and terms. However, a new feature is now available. To 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 :

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

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

This rule actually splits into several derivation rules added to the typed system:

[bar] $$\frac{A \text{ type}}{\overline{A} \text{ type}} \qquad \frac{t : A}{t : \overline{A}} \qquad \frac{t : \overline{A} \qquad t{\downarrow}}{t : A}$$

Let us give some examples for the formation of types in the extended typed system $\mathcal{T}^+$:

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

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

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

They 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.

Introduction of fixpoints 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 [6, 7] and in [21] where a consequent study of this kind of extended typed system is done in the NuPrl framework.

2.4 Application for these ideas within NuPrl

We give a brief overview for the main results and problems encountered in the adaptation of the general idea in NuPrl type system. We expect the following lines to be not too much reducing with respect to [21]. NuPrl is a computational typed theory, where pure terms are given types using a Damas-Milner like algorithm. Therefore, extending such a framework to deal with fixpoints is easy a priori. The approach is twofold. First, pure term is considered, and its possible convergence is checked. It is even possible to develop a theory of convergence [2], based on evaluation trees and extending Böhm trees model of solvable terms presented in [3]. The key point is a continuity property required for linking computational behavior of a fixpoint to that of its approximations.

The second step is thus expected to provide a type for the eventually convergent term. This point appears to raise serious difficulties. The same kind of property is required, but now over type expressions. We need the fact the type given to a fixpoint can be continuously connected to types progressively given to the approximations. The problem is briefly explained in [2]. However no serious effort have been made to give a solution. A severe restriction is imposed in [21], which mainly relies on the presence of propositional assertions in NuPrl.

Therefore, we devote our attention to the adaptation of this idea to a somewhat different typed system. The choice of the Calculus of Constructions comes from the fact none of the problems encountered within NuPrl appear; mainly, terms are strongly typed and there is no propositional assertions since the Curry-Howard isomorphism is not taken as an identity.

2.5 Adaptation to the Calculus of Constructions

We first provide a very short review of the Calculus of Constructions. Thus main notations and the basic inference system is introduced, together with some important aspects from the point of view of the discussion for the way we dealt with the adaptation for Constable and Smith' ideas in this quite different framework.

A recap on the Calculus of Constructions

The Calculus of Constructions CC is a higher order lambda-typed calculus which allows, through the Curry-Howard isomorphism, the representation of proofs form the higher order Intuitionistic logic. However, the isomorphism between propositions and types is not taken as an identification. Thus one finds a type of "propositions", noted **Prop**, and itself a type of type **Type**. The different kinds of data types are coded owing to the system impredicativity. Type dependency, coming form Martin-Löf Intuitionistic Type Theory, allows the coding for refined logical propositions. Concretely, terms and types are mixed in the syntax, using $(M\ M)$ for application, $[x : M]M$ for lambda-abstraction and $(x : M)M$ as a notation for dependent type. The calculus enjoys confluence and strong normalization for the usual β-reduction. The rules of inference are given in table 1.

Owing to these inference rules, it is possible to define invariants on well formed terms in CC. The distinction between terms and types lies on the following invariants. A well formed term M is a **propositional type** if M : **Type** and **propositional schema** if $M : A$, with A a propositional type; M is a proposition if M : **Prop** and a **proof** if $M : A$ with A a proposition. Notice a proposition is a propositional schema, since **Prop** : **Type**. These notions are well defined since they are actually invariant under type conversion. A **type** is then either a propositional type or a proposition. An **object** is any well formed term which is not a propositional type. See [4, 17] for a complete presentation.

Some informal hints

Although the adaptation from NuPrl to CC followed a somewhat sinuous way, it is possible to explain the translation shortly. We are about to work with terms for which termination can no longer be guaranteed. This is what we want at the level of programs. Nevertheless, logical proofs and programs are both represented by the same objects, the same typed terms in the system. But we do want to consider that a logical proof must be "complete" in some sense. So, from the logical point of view, strong normalisation should be a important property to be preserved as far as possible.

Thus, it seems clear 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 though 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 part 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 [17], Paulin-Mohring introduced a new constant **Spec** (**Set** elsewhere) in order to distinguish between terms with or without any informative content. The purpose is there to separate pure logical proofs from proofs having some computational content, and thus providing programs through extraction process. And such a trick works, since then, terms are in an unambiguous correspondence with exactly one of the two constants **Prop** and **Spec**. Our solution follows the same idea. We introduce a new constant, denoted $\overline{\mathbf{Prop}}$, in order to mark terms to be considered as partial terms. This is for the construction of bar types in the Calculus of Constructions. Partial programs and partial schemes 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 is taken into account through the predicate of convergence within NuPrl. We consider that this predicate cannot be put in the system ; but rather belongs to a metalevel with respect to CC system. Actually, either t diverges and so predicate $t\downarrow$ cannot answer, or t converges and $t\downarrow$ terminates with true. Hence this predicate carries a useless information. Moreover, the logical information contained in $t\downarrow$ cannot be taken as an object on which we are able to reason with, since we do not want to introduce propositional assertions: the problems raised in such a situation have been invoked above. So this part will be dropped out from our extension.

3 Presentation of CC^+ and main results

This section gives a short overview of CC^+. We refer to [1, 2] for a complete presentation.

3.1 Preterms, reductions, rules

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

The set of preterms of CC^+, is the least set of terms containing the constants **Prop** and $\overline{\mathbf{Prop}}$, a denumerable set $\mathcal{V}$ of variables, and generated by the following grammar:

$$M \;::=\; x \;\mid\; \mathbf{s} \;\mid\; (M\ M) \;\mid\; [x:M]M \;\mid\; (x:M)M \;\mid\; \langle x:M\rangle M$$

where $x \in \mathcal{V}$, $\mathbf{s} \in \{\mathbf{Prop}, \overline{\mathbf{Prop}}\}$ and the new notation $\langle x:M\rangle M$ is introduced to denote a **fixpoint**.

The definition of substitution is straightforward. Besides the usual β-reduction, a new reduction is introduced, which deals with fixpoint unfolding:

$$\boldsymbol{\beta}\mathbf{fix} \qquad \langle x:M\rangle N \longrightarrow_{\beta fix} ([x:M]N \;\; \langle x:M\rangle N)$$

where M,N, L are preterms, and $x \in \mathcal{V}$. Let $\rightarrow^*_+$ for the reflexive and transitive closure of $+ \equiv \beta \cup \beta fix$ and $\simeq$ 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. Because we deal with fixpoint computations we must take *head* reductions as meaningful.

Table 2 gives the additional rules required for the extension CC^+. Letter $\mathbf{s}$ stands for one or other of the constants **Prop**, **Type**, $\overline{\mathbf{Prop}}$ and $\overline{\mathbf{Type}}$. As already pointed out, fixpoints can be introduced at both proof and schema level. Table 2 shows a distinction between the two kinds of introductions. At the level of proofs, we get recursive programs, as desired. Fixpoints at the schema level makes it possible to extend the type system with respect to specifications allowed to be built. To some extent, this feature is more than want we wanted initially. Section 4 will provide a illustration for this part of the extension. We now explain why it has been necessary to restrict seriously the recursive schemas allowed to be introduced via fixpoints.

3.2 Positive occurrences of a variable

Let us consider the schema $X : \overline{\mathbf{Prop}} \vdash X \to X : \overline{\mathbf{Prop}}$. With no restriction, it is possible to consider the proposition $\Lambda \equiv \langle X : \overline{\mathbf{Prop}} \rangle X \to X : \overline{\mathbf{Prop}}$. Then we get the equality $\Lambda \simeq \Lambda \to \Lambda$. Thus we can form the terms $\delta \equiv [x : \Lambda](x\ x) : \Lambda$ and $\Omega \equiv (\delta\ \delta) : \Lambda$. Clearly normalization with respect to β -reductions is definitely lost. Therefore, a necessary condition is that X appears with "positive" occurrences in the schema. This is a quite usual condition which moreover reveals to be sufficient to keep the calculus strongly normalizing under β-reductions. And this is the best property we could expect in presence of fixpoints. Now, let us give the formal definition for this syntactic restriction on terms.

For that purpose, let us introduce new definitions for preterms. The sets of **positive** and **negative** preterms with respect to a fixed variable X are defined by the following grammar:

$$\begin{array}{rcl} Pos & ::= & M \mid X \mid (Pos\ m) \mid [x : M]Pos \mid (x : Neg)Pos \\ Neg & ::= & M \mid (Neg\ m) \mid [x : M]Neg \mid (x : Pos)Neg \end{array}$$

with the restriction that X does not occur free in M or m. The notation $Pos_X(T)$ (resp. $Neg_X(T)$) indicates T is positive (resp. negative) with respect to X.

It is impossible to assume the previous definitions to be given on normal terms, for instance, since properties such as normalization relies on this restriction scheme. This is the reason why they appear to be so drastic, in contrast to the solution proposed in [8]. For the same reason, we have to provide a definition for which the predicates Pos and Neg are invariant under conversion. Therefore, the positivity condition for a fixpoint results from the cases for the application and the abstraction, due to the βfix-reduction rule. We must require $Pos_X(\langle x : P \rangle M)$ (resp. $Neg_X(\langle x : P \rangle M)$) iff $X \notin FV(\langle x : P \rangle M)$.

3.3 Main metaresults

It is interesting to note the introduction of fixpoints does not introduce too much trouble in the properties one expects from a typed system. Actually, we get the best we can expect:

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

uniform properties The following properties are satisfied 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 [1] ; for a complete treatment see [2].

Now let us make precise how there could exist a "canonical" marking among terms and how the property for a term to be in normal form can be extended in a reasonable way.

Partial and total terms Let M be a well typed term with type N. We shall say that M is **total** if either $N \equiv \mathbf{Type}$ or N is of type **Type** or **Prop**. Otherwise such a term is said to be **partial**. We respectively denote $\mathsf{CC}^+{}_{tot}$ and $\mathsf{CC}^+{}_{par}$ the two parts 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 2.5.

"Almost in normal form" terms The set of normal forms (NF) is too restrictive. For example, the term $[x : P]x$ where $P \equiv \langle X : \overline{\mathbf{Prop}}\rangle(C : \overline{\mathbf{Prop}})(X \rightarrow C) \rightarrow C$ has no derivation to a normal form. However, the type information provided by the "abstraction" part of a term has no computational meaning ; it is forgotten in the pure lambda-term obtained through the erasure function.

Thus we have to propose a set of terms weaker than NF (normal forms). Briefly sketched, it suffices to define a term M to be in ANF (almost normal forms) if the term $\mathcal{E}(M)$ obtained form M by through $\mathcal{E}(\{x : P\}M) = \mathcal{E}(M)$ and $\mathcal{E}(M\ N) = (\mathcal{E}(M)\ \mathcal{E}(N))$ is in normal form. Clearly, NF $\subset$ ANF $\subset$ HNF, where HNF stands for the set of head normal forms.

4 Internal codings for integers

This part is devoted to the question how both computational and logical expressiveness are possibly improved in CC^+. The main originality of this extension is the ability to give alternative codings for integers. A complete treatment is already given in [1, 2] for CC^+, and in [12, 13] 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 give some intuitive understanding for problems encountered when trying to translate mathematical notions such as different forms of equality, equivalences or isomorphisms.

4.1 Primitive recursion over integers

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

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

Informally :

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

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

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

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

4.2 Integers as iterators

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

$$\mathbf{Nat}^i \equiv (C : \mathbf{Prop})C \to (C \to C) \to C : \mathbf{Prop}$$

Thus this representation mimics as well as possible the position of $\mathbf{Nat}^i$ as a nno in the typed system. Let O^i and S^i be the closed terms which code zero and the successor function, respectively. We get:

$$
\begin{array}{rcl}
\mathbf{O}^i & \equiv & [C : \mathbf{Prop}][x : C][f : C \to C]x \\
\mathbf{S}^i & \equiv & [n : \mathbf{Nat}^i][C : \mathbf{Prop}][x : C][f : C \to C](f\ (n\ C\ x\ f))
\end{array}
$$

Then the "regular integers" are coded through the set $\{((\mathbf{S}^i)^k\ \mathbf{O}^i) \mid k \in \mathbf{N}\}$. Let us have a look at expressiveness issues.

Logical expressiveness

The induction predicate is:

$$\mathbf{Ind}^i \equiv [n : \mathbf{Nat}^i](P : \mathbf{Nat}^i \to \mathbf{Prop})(P\ \mathbf{O}^i) \to ((n : \mathbf{Nat}^i)(P\ n) \to (P\ (\mathbf{S}^i\ n))) \to (P\ n)$$

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. No proof exists for $(n : \mathbf{Nat}^i)(\mathbf{Ind}^i\ n)$. As noticed in [15, 17], we need the fact that $(n\ \mathbf{Nat}^i\ \mathbf{O}^i\ \mathbf{S}^i)$ equals n. But this property means precisely that every $n : \mathbf{Nat}^i$ 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, as proved as meta result.

Computational expressiveness

A solution to primitive recursion schema is provided in CC by $g \equiv [n : \mathbf{Nat}^i](\mathbf{Rec}^i\ n\ A\ a\ h)$, where $\mathbf{Rec}^i$ is the recursor term, and is defined as follows:

$$\begin{aligned}
\mathbf{Rec}^i \ &\equiv\ [n : \mathbf{Nat}^i][C : \mathbf{Prop}][x : C][h : \mathbf{Nat}^i \times C \to C] \\
&\qquad (\mathbf{snd}\ (n\ \ \mathbf{Nat}^i \times C\ \ (\mathbf{pair}\ \mathbf{O}^i\ x)\ [y : \mathbf{Nat}^i \times C](\mathbf{pair}\ (\mathbf{S}^i\ (\mathbf{fst}\ y))\ (h\ y)))) \\
&:\ \mathbf{Nat}^i \to (C : \mathbf{Prop})C \to (\mathbf{Nat}^i \times C \to C) \to C
\end{aligned}$$

This coding leads to the following computations [16]:

$$\begin{aligned}
(\mathbf{Rec}^i\ \mathbf{O}^i\ C\ x\ h) &\simeq x \\
(\mathbf{Rec}^i\ (\mathbf{S}^i\ n)\ C\ x\ h) &\simeq (h\ (\mathbf{pair}\ (\mathbf{Rec}^i\ n\ C\ x\ h)\ f_n))
\end{aligned}$$

with $f_{\mathbf{O}^i} \simeq \mathbf{O}^i$ and $f_{(\mathbf{S}^i\ n)} \simeq (\mathbf{S}^i\ f_n)$.

Nevertheless the equality $f = id_{\mathbf{Nat}^i}$ can be proved only for 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 [17, section 4.4.1].

Assume we are dealing with those terms representing regular integers, and let us give a more concrete illustration for this problem, by considering 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

$$\begin{tikzcd}
n \arrow[r] \arrow[d] & n+1 \arrow[d] \\
\langle f(n), p(n) \rangle \arrow[r] & \langle f(n+1), n \rangle
\end{tikzcd}$$

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 : they 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 [5, 20, 19] a solution is given, through an extension of CC with inductive types. Our solution is similar to recursive coding introduced in [12, 13] through an extension of $\mathsf{AF2}$, called TTR.

4.3 Integers as selectors

In a cartesian closed category, any nno $\mathbf{N}$ satisfies another diagram :

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

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

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

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

Actually $\mathbf{Nat}^i$ satisfies this kind of diagram too, at least when we restrict ourselves to the set of codes for regular integers. But then, we rather get a weak form of equivalence through the existence of two terms such that :

$$\mathbf{Nat}^i \underset{\mathbf{in}}{\overset{\mathbf{out}}{\rightleftharpoons}} (C : \mathbf{Prop})C \to (\mathbf{Nat}^i \to C) \to C$$

such that $\mathbf{in} \circ \mathbf{out} = id$ and $\mathbf{out} \circ \mathbf{in} = id$.

However, using fixpoint at the level of schemas, we are actually able to define $\mathbf{Nat}^r$ such that

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

It suffices to define

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

Then we get

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

where $\mathbf{P}^r$ is a term for the predecessor function.

Although, the case of the predecessor does not use the primitive recursion schema in its generality, we are able to give a solution to any primitive recursion schema, using fixpoint at the level of programs :

$$g = \langle x : \mathbf{Nat}^r \to A\rangle [n : \mathbf{Nat}^r](n \quad A \quad a \quad [p : \mathbf{Nat}^r](h \quad p \quad (x \quad p)))$$

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. It is also possible to give a solution for μ-recursion schema : given $\phi : \mathbf{N} \to \mathbf{N}$ represented by the term $f : \mathbf{Nat}^r \to \mathbf{Nat}^r$, find the least integer n such that $\phi(n) = 0$. 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 $(G\ \mathbf{O^r})$ where

$$G \equiv \langle H : \mathbf{Nat}^r \to \mathbf{Nat}^r\rangle [k : \mathbf{Nat}^r](f\ k\ \mathbf{Nat}^r\ k\ [p : \mathbf{Nat}^r](H\ (\mathbf{S^r}\ k))$$

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

- $\forall n \in \mathbf{N}\ (\mathbf{P^r}\ (\mathbf{S^r}\ \underline{\mathbf{n}})) \to^* \underline{\mathbf{n}}$ (in a fixed number of step indeed);
- The set of closed ANF terms of type $\mathbf{Nat}^r$ is the set $\{\underline{n} \mid n \in \mathbf{N}\}$;
- All partial recursive functions over integers are definable in CC^+. (Hint : the proof follows the same lines as Barendregt's in [3])

Logical expressiveness

We pointed out $\mathbf{Nat}^i$ behaves as well as a nno provided it is restricted to the subset of codes for regular integers. Since then $\mathbf{Nat}^i$ satisfies the second diagram, there must exist a correspondence between $\mathbf{Nat}^i$ and $\mathbf{Nat}^r$. And it is actually the case through the terms :

$$\begin{array}{rcl} i & \equiv & [n:\mathbf{Nat}^i](n\ \mathbf{Nat}^r\ \mathbf{O}^r\ \mathbf{S}^r) : \mathbf{Nat}^i \to \mathbf{Nat}^r \\ r & \equiv & [n:\mathbf{Nat}^r](n\ \mathbf{Nat}^i\ \mathbf{O}^i\ [p:\mathbf{Nat}^r](\mathbf{S}^i\ (r\ p))) : \mathbf{Nat}^r \to \mathbf{Nat}^i \end{array}$$

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

(i) $(n:\mathbf{Nat}^i)(\mathbf{Ind}^i\ n) \to (\mathbf{Ind}^r\ (i\ n))$ and $(n:\mathbf{Nat}^i)(\mathbf{Ind}^i\ n) \to n = (r \circ i\ n)$.

(ii) $(n:\mathbf{Nat}^r)(\mathbf{Ind}^r\ n) \to (\mathbf{Ind}^i\ (r\ n))$ and $(n:\mathbf{Nat}^r)(\mathbf{Ind}^r\ n) \to n = (i \circ r\ n)$.

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. so there is no reason why CC^+ should give us a way to prove all Peano axioms.

4.4 Yet another coding

As already shown in [12], it is possible to give an alternate coding for integers, mixing both iterative and selective feature in a unique recursive definition:

$$\mathbf{Nat}^* \equiv \langle X:\overline{\mathbf{Prop}}\rangle(C:\overline{\mathbf{Prop}})C \to (C \to X \to C) \to C : \overline{\mathbf{Prop}}$$

For this coding, we get:

$$\begin{array}{rcl} \mathbf{O}^* & \equiv & [C:\overline{\mathbf{Prop}}][x:C][f:C \to \mathbf{Nat}^* \to C]x \\ \mathbf{S}^* & \equiv & [n:\mathbf{Nat}^*][C:\overline{\mathbf{Prop}}][x:C][f:C \to \mathbf{Nat}^* \to C](f\ (n\ C\ x\ f)\ n) \\ \mathbf{P}^* & \equiv & [n:\mathbf{Nat}^*](n\ Nat^*\ \mathbf{O}^*\ \mathbf{snd}) \end{array}$$

Now the primitive recursion schema can be given a solution with no need for a fixpoint. Actually, it suffices to write:

$$\mathbf{Rec}^* \equiv [n:\mathbf{Nat}^*][C:\overline{\mathbf{Prop}}][x:C][h:C \to \mathbf{Nat}^* \to C](n\ C\ x\ h) \equiv id_{\mathbf{Nat}^*}$$

Since, then, the equations

$$\begin{array}{rcl} (\mathbf{O}^*\ C\ x\ h) & \simeq & x \\ ((\mathbf{S}^*\ n)\ C\ x\ h) & \simeq & (h\ (n\ C\ x\ h)\ n) \end{array}$$

are trivially satisfied. The solution is quite the same as the recursor introduced in [17] or in [19] when translating a fraction of the Calculus of Constructions with Inductive definitions into CC^+. Notice, however, this coding fails to allow proving principle of induction.

5 Conclusion and further developments

We restricted our study to the data type of integers through different internal codings. However, the example is sufficiently representative for all the enhancements and limitations we can expect form the extension of CC with fixpoints, as presented in [1, 2].

Owing to the very nature of the Calculus of Constructions, the introduction for recursive features has shown to be more uniform than in the AF2 framework [13]: in CC, we have only to consider fixpoint for a functional term, as well for proof-terms as for schema-terms. However, it seems hard to pursue the comparison longer, since the former develops through semantic arguments, where CC and CC^+ are very syntactic in nature.

The introduction of recursive schemas appears as the most interesting part of the extension, since then we are able to propose alternative codings for data structures and logical specifications, thus providing (extracted) programs with better intentional behavior, although the extensional expressiveness of the system was not expected to be increased. Actually, provably total numerical functions are already represented in CC. Morever, through correspondence between the different codings, we showed how logical expressiveness cannot be improved, for the consideration of the induction principle, for instance. However, this point is not the only valuable criteria, if we address logical expressiveness issue. Remind, our initial motivation was to add fixpoint constructs into a typed framework in order to understand how it is possible to reason about them, hence giving a logical validation for programs allowing recursive features. The relationship between inductive and recursive coding for integers gives a quite interesting, although partial, answer to that question. Actually the question is that of founding a valuable methodology for programs development.

The methodology can be explained in the schematic form which follows: let T^i be a *inductive* specification formalized in pure Calculus of Constructions and T^r for the corresponding *recursive* specification written with the help of fixpoint(s). Then it is possible to translate a proof-term m^i for T^i into a (recursive) proof-term m^r for T^r. This method respects our initial motivation for introduction of fixpoints, and insures the (head) computation of m^r with terminates. And indeed, this is all we need. Notice this argument makes sense for data types. However, in case the specification is too involved, this methodology applies as soon we consider the extracted programs. See introduction of [2], Paulin and Werner [18].

Hence, we are suggested to investigate the study of recursivity form the opposite way round. Starting from a logical basis, where recursivity is introduced through logical justification and understanding. Thus, recursive proof-terms should be considered only in the programming part of the system, either in the extraction process, or as an extended feature of $F\omega$ component in the same way as in [17]. To this respect, Paulin's suggestion for considering a weaker extension than CC^+ where fixpoints are removed at the level of proofs, preserved at the level of schemas deserves further attention.

Acknowlegements

I am pleased to thank G. Huet who offered me the opportunity to present this work at the '92 workshop of the BRA "Types for Proofs and Programs", in Båstad, and to C. Paulin and L. Bougé, for their remarks on previous versions of this paper.

References

[1] Audebaud P. (1991) Partial Objects in the Calculus of Constructions. In *6nd Conf. on Logic in Comp. Science*. IEEE, 1991.

[2] Audebaud P. (1992) Extension du Calcul des Constructions par points fixes. Thèse, Université Bordeaux I, 1992. Available through anonymous ftp to lip.ens-lyon.fr in /pub/LIP/users/paudebau repertory.

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

[4] Coquand T. and Huet G. (1988) The Calculus of Constructions. *Inf. Comp.* **76** p.95-120.

[5] Coquand T. and Paulin-Mohring P. (1990) Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*. LNCS **417**, Springer Verlag 1990.

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

[7] 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.

[8] 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.

[9] Crole R.L. and Pitts A.M. (1990) New foundations for Fixpoint Computations. In *Proceedings of the 5th Conference on Logic In Computer Science. IEEE 1990.*

[10] 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.

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

[12] 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. 1988.

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

[14] Parigot M. (1992) $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings International Conference on Logic Programming and Automated Deduction, St Petersburg.* LNCS **624**. Springer Verlag. 1992.

[15] Paulin-Mohring C. (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.

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

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

[18] Paulin-Mohring C. and Werner B. (1993) Synthesis of ML programs in the system Coq. J. Symbolic Computation 1993, **11**, pp 1-34.

[19] Paulin-Mohring Ch. (1993) Inductive definitions in the system Coq. Rules and properties. In *Proceedings of the 1st Conference on Typed Lambda Calculi and Applications, Ultrecht (eds. Bezem M., Groote J.-F.)* LNCS **664**. Springer Verlag. 1993.

[20] Pfenning F. and Paulin-Mohring C. (1990) Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics.* LNCS 442. Springer Verlag. 1990.

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

empty	$$\frac{}{[\,] \vdash}$$
val-intro	$$\frac{\Delta \vdash P : \mathbf{s} \qquad x \in \mathcal{V} \qquad x \notin \Delta}{\Delta, x : P \vdash}$$
hypo	$$\frac{\Delta \vdash \qquad x : P \in \Delta}{\Delta \vdash x : P}$$
Prop-intro	$$\frac{\Delta \vdash}{\Delta \vdash \mathbf{Prop} : \mathbf{Type}}$$
prod-intro	$$\frac{\Delta, x : P \vdash Q : \mathbf{s}}{\Delta \vdash (x : P)Q : \mathbf{s}}$$
abs-intro	$$\frac{\Delta, x : P \vdash M : Q}{\Delta \vdash [x : P]M : (x : P)Q}$$
apply	$$\frac{\Delta \vdash M : (x : P)Q \qquad \Delta \vdash N : P}{\Delta \vdash (M\ N) : Q[x/N]}$$
type-conv	$$\frac{\Delta \vdash M : P \qquad P \simeq Q}{\Delta \vdash M : Q}$$

Table 1: Rules of inference for CC

$\overline{\textbf{Prop}}$**-intro**	$\dfrac{\Delta \vdash}{\Delta \vdash \overline{\textbf{Prop}} : \overline{\overline{\textbf{Type}}}}$
fix-in-$\overline{\textbf{Prop}}$	$\dfrac{\Delta \vdash P : \overline{\textbf{Prop}} \qquad \Delta, x : P \vdash M : P}{\Delta \vdash \langle x : P \rangle M : P}$
fix-in-$\overline{\textbf{Type}}$	$\dfrac{\Delta \vdash P : \overline{\textbf{Type}} \qquad \Delta, x : P \vdash M : P \qquad Pos_x(M)}{\Delta \vdash \langle x : P \rangle M : P}$
eq-$\boldsymbol{\beta}$**-fix**	$\dfrac{\Delta \vdash \langle x : P \rangle M : P}{\Delta \vdash \langle x : P \rangle M = ([x : P]M \ \langle x : P \rangle M)}$

Table 2: Additional rules for CC^+

A Symmetric Lambda Calculus for "Classical" Program Extraction

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

e-mail: { barba,stefano} @di.unito.i

DRAFT

Abstract

In the present paper we introduce a λ-calculus with symmetric reduction rules and "classical" types, i.e. types corresponding to formulas of classical propositional logic. Strong normalization property is proved to hold for such a calculus. We then extend this calculus in order to get a system equivalent to Peano Arithmetic and show, by means of a theorem on the shape of terms in normal form, how to get recursive functions out of proofs of Π_2^0 formulas, i.e. the ones corresponding to program specifications.

1 Introduction

The possibility of extracting recursive functions out of intuitionistic proofs expressing their specifications, i.e., in general, the *effective* features of constructive mathematics, has had a leading rôle in the development of computer science. This rôle, not limited only to foundational aspects, has been played in a wide field of research, with the aim of supporting the *working* computer scientist. In particular, the correspondence between logical constructive systems and several λ-calculi, known as the Curry-Howard analogy, has been used to develop prototypes of systems for the design and development of provably correct programs [Con 86] [NPS 90] [PN 90].
Classical logics was always left out of the particular field of research relating logics and type-theories. This, however, has not to be imputed to its complete lack of effective features. Indeed, as far as the part of logics relevant for computer science is concerned, this is absolutely not the case. Quite old and well known theoretical results (for instance [Kre 58]) makes it sure that it is possible to transform a *classical* proof of $\forall x \exists y.P(x,y)$ (with P decidable) into a *recursive* function f such that, for any x, $P(x,f(x))$ holds. This means that it has been for long time possible to syntesize, out of a classical proof that a specification is satisfiable, a program that satisfies this specification. What was still preventing classical logics to have a more relevant rôle in computer science, was instead the lack of *clear* and *practical* methods to extract constructive content from classical proofs, and systems which helped to understand their constructive features.
Lots of efforts have begun to be made, in the last few years, in this direction. Among them it has

to be mentioned the interpretation of classical logics into calculi with continuations ([Gri 90], [Mur 90]), originating from Friedman's A-translation ([Fri 78]). An investigation of Prawitz's set of reductions for classical logic ([Pra 65], [Pra 81]) was instead the starting point of a method to extract constructive content from classical proofs devised in [BB 91], which has an interpretation in terms of a valuation semantics [BB 92].
All the above mentioned methods have natural deduction versions of classical logics at their roots. On the "sequent calculus-side" reseach efforts have led to the $\lambda\mu$-calculus of Parigot [Par 92] and to Coquand's *game-theoretical* interpretation [Coq 92].

These preliminary results, at least the ones on the "natural deduction-side", share a common problem: that of complex sintax. This, of course, represents a serious obstacle towards a neat and full understanding of classical logics from a computational point of view.
Our main aim in the present paper is then that of defining a system for "classical program extraction" which is simple enough. Our starting point, in Section 2, will be defining a "classical" simply typed λ-calculus (λ^{Sym}_{Prop}), i.e. a λ-calculus which is in a Formulas-as-Types correspondence with propositional classical logics. In this system negation is not a primitive connective, and we manage to identify a type (formula) A with its double negation $A^{\perp\perp}$. This enables to get a system where we have a *symmetric* application, such that either component of an application can be virtually looked at, indifferently, as function or argument. Because of this symmetry all the reductions of the calculus have a dual version. It is relevant to stress that the reductions we define are simple and natural ones, and, differently from what was done in other systems for "classical program extraction", no particular reduction is introduced because of the particular use we intend to make with such a system. Our system λ^{Sym}_{Prop} is then proved in Section 5 to be strongly normalizable using a non trivial version of Tait-Girard's computability method: symmetric candidates. In Section 3 system λ^{Sym}_{Prop} is extended with first order features in order to obtain a system corresponding to Peano Arithmetic ($\lambda^{Sym}_{\mathbf{PA}}$). By means of a Shape of Normal Forms Theorem, proved in Section 4, it is then possible to extract the constructive content of terms corresponding to proofs of formulas of the form $\forall x \exists y.P(x,y)$ with P decidable.

2 λ^{Sym}_{Prop}: A Symmetric Simply Typed "Classical" λ-calculus

In this section we introduce the system λ^{Sym}_{Prop}. In such a system types correspond to formulas and terms to proofs of propositional classical logic. We shall then often use indifferently the words *type, formula, proposition* and *term, proof.*

The basis to build the types of our system consists in two sets of base types: $\mathcal{A} = \{a, b, \ldots\}$ (atomic types) and $\mathcal{A}^{\perp} = \{a^{\perp}, b^{\perp}, \ldots\}$ (negated atomic types). Out of these two sets we build, as shown below, *minimal-types* and *types.*

Definition 2.1 *(i) The set of* minimal-types (m-types *for short) is defined by the following grammar:*

$$A ::= \alpha \mid \alpha^{\perp} \mid A \wedge A \mid A \vee A$$

where α ranges over $\mathcal{A}$ and $\alpha^{\perp}$ over $\mathcal{A}^{\perp}$.

(ii) The set of types *is defined by the following grammar.*

$$C ::= A \mid \perp$$

We need to define first the m-types since we wish to have a calculus where formulas does not contain the absurdity proposition as proper subtype. Such a choice is motivated by technical reasons, as it will be clearer in Section 5. It is easy to check however that this is no restriction at all (a formula $A \wedge \bot$ can always be identified with $\bot$, and $A \vee \bot$ with A).

In the following we shall denote types by $A, B, C, D, A_1, A_2, C_1, C_2 \ldots$.

By having a set of atomic types and a set of negated atomic types, it is easy to see that we have a propositional calculus where negation is neither primitive nor defined in terms of $\bot$.

Definition 2.2 *We define the negation $A^\perp$ of a type A as follows:*

1. $(\alpha)^\perp = \alpha^\perp$	*2.* $(\alpha^\perp)^\perp = \alpha$
3. $(A \wedge B)^\perp = A^\perp \vee B^\perp$	*4.* $(A \vee B)^\perp = A^\perp \wedge B^\perp$

We get then a calculus with involutive negation.

Lemma 2.1

$$A^{\perp\perp} = A.$$

Proof. By induction on A, using Def. 2.2.□

Definition 2.3 (λ^{Sym}_{Prop}-rules) *The terms of the system λ^{Sym}_{Prop} are defined by the following rules :*

$$var)\ \frac{}{x^A : A}$$

$$\langle,\rangle)\ \frac{P_1 : A_1 \quad P_2 : A_2}{\langle P_1, P_2\rangle : A_1 \wedge A_2} \qquad \sigma_i)\ \frac{P_i : A_i}{\sigma_i^{A_1,A_2}(P_i) : A_1 \vee A_2}(i = 1, 2)$$

$$\lambda)\ \frac{\begin{array}{c}[x : A]\\ \vdots\\ P : \bot\end{array}}{\lambda x.P : A^\perp} \qquad \star)\ \frac{P_1 : A^\perp \quad P_2 : A}{(P_1 \star P_2) : \bot}$$

In the following the type of a term will be often denoted by superscripts while the superscripts A_1, A_2 in terms like $\sigma_i^{A_1,A_2}(P_i)$ will be often omitted.

Remark 2.1 *The propositional classical logic associated to our system is complete. Rules and connectives not given above can be derived as it is usual in classical logic. We show below the (type part) of the derivation of the conjunction-elimination rule and the implication-elimination rule.*

$$\cfrac{\cfrac{A_1 \wedge A_2 \equiv (A_1^\perp \vee A_2^\perp)^\perp \qquad \cfrac{[A_i^\perp]}{A_1^\perp \vee A_2^\perp}}{\bot}}{A_i}$$

$$A \to B =_{Def} A^\perp \vee B$$

$$\dfrac{A \to B \equiv A^{\perp} \vee B \qquad \dfrac{A \qquad [B^{\perp}]}{(A^{\perp} \vee B)^{\perp} \equiv A \wedge B^{\perp}}}{\dfrac{\perp}{B}}$$

We call the operator "$\star$" *symmetric application* [1] , since, given the terms $P^{A^{\perp}}$ and Q^A, both $P^{A^{\perp}}\star Q^A$ and $Q^A\star P^{A^{\perp}}$ are correct λ^{Sym}_{Prop}-terms. This symmetry is reflected by the (pairwise dual, but rule $(Triv)$) reductions rules defined below.

Definition 2.4 (λ^{Sym}_{Prop}-reduction rules)

$$\left\{\begin{array}{llll} \beta) & \lambda x.P\star Q & \to_{\beta} & P[Q/x] \\ \beta^{\perp}) & Q\star\lambda x.P & \to_{\beta^{\perp}} & P[Q/x] \end{array}\right.$$

$$\left\{\begin{array}{lllll} \eta) & \lambda x.(P\star x) & \to_{\eta} & P & (^1) \\ \eta^{\perp}) & \lambda x.(x\star P) & \to_{\eta^{\perp}} & P & (^1) \end{array}\right.$$

$$\left\{\begin{array}{lllll} \pi) & \langle P_1,P_2\rangle\star\sigma_i(Q_i) & \to_{\pi} & P_i\star Q_i & (i=1,2) \\ \pi^{\perp}) & \sigma_i(Q_i)\star\langle P_1,P_2\rangle & \to_{\pi^{\perp}} & Q_i\star P_i & (i=1,2) \end{array}\right.$$

$$Triv)\ E[P] \quad \to_{Triv} P \quad (^2)$$

$(^1)$ *if $x \notin FV(P)$.*
$(^2)$ *if $E[-]$ is a context $\neq [-]$ with type $\perp$, P has type $\perp$ and $FV(P) \subseteq FV(E[P])$*

In the following $\to_1$ will denote the union of the reduction relations defined above. $\to$ will denote the reflexive and transitive closure of $\to_1$.

Remark 2.2 *Rule (π), which is inspired by the sequent calculus, looks as follows in terms of derivations:*

$$\dfrac{\dfrac{A_1 \quad A_2}{A_1 \wedge A_2} \qquad \dfrac{A_i^{\perp}}{A_1^{\perp} \vee A_2^{\perp}}}{\perp} \qquad \leadsto \qquad \dfrac{A_i \quad A_i^{\perp}}{\perp}$$

This rule in our system plays the rôle of the usual reduction rule for getting rid, in natural deduction, of an introduction of a conjuction followed by its elimination. As we have seen in remark 2.1, the elimination of conjunction can be derived in our system. It is then easy to see that, by using that derived rule, the usual reduction rule can be defined in terms of one reduction (π) and one (η).
Notice that, by defining the (π) rule as we have done, we manage easily to dualize *it.*

Definition 2.5 *Let k be an integer and P a term.*

(i) k is a bound *for P if the reduction tree of P has a finite height $\leq k$.*

(ii) P strongly normalizes *if it has a bound.*

[1] Note that instead of defining a symmetric application, a possibility could have been that of defining a calculus where the terms $P\star Q$ and $Q\star P$ are identified, i.e. a calculus where terms are indeed equivalence classes. This however, besides being less intuitive, would have led outside the notion of formal system.

Then a term strongly normalizes iff its reduction tree is finite.
This definition has to be preferred to the usual one, i.e. "each reduction sequence out of P is finite", because the latter is intuitionistically weaker than the former (classically, they are equivalent through König's Lemma).

Notation. The following notations will be used.

$$\begin{array}{lcl} Var_C & = & \{\text{variables of type } C\} \\ Term_C & = & \{\text{terms of type } C\} \\ SN_C & = & \{P \in Term_C \mid P \text{ strongly normalizes}\} \end{array}$$

One of the main properties enjoyed by system λ^{Sym}_{Prop} which will be essential for its application is that of strong normalization.

Theorem 2.1 (Strong Normalization for λ^{Sym}_{Prop}) *Let C be a type.*

$$Term_C = SN_C.$$

The proof of this theorem will be the argument of Section 5.

It is no surprise that , as many proposed calculi dealing for classical logic, λ^{Sym}_{Prop} does not have the Church-Rosser property. This it is easy to check by considering that we can have terms of the form $(\lambda x.P)\star(\lambda y.Q)$ which can lead, by means of reductions (β) and $(\beta^{\perp})$, to two different normal forms.

3 $\lambda^{Sym}_{\mathbf{PA}}$: A Calculus for Peano Arithmetic

In Section 2 we have defined a calculus based on a version of propositional classical logic. This logic, even if it can be considered as a possible basis for a (simply) typed λ-calculus with symmetric application and *classical* types, is however too poor for our present purposes, i.e. the investigation of the computational content of classical reasoning. In fact we defined it only as starting point.
We wish a logic in which it is possible to express and prove specifications of programs and therefore a first order logic. We make then the choice of Peano Arithmetic. In the following we shall define a calculus corresponding to a natural deduction version of Peano Arithmetic and based on system λ^{Sym}_{Prop}. We shall call such a calculus $\lambda^{Sym}_{\mathbf{PA}}$.

We begin by defining Peano Arithmetic terms (PA-terms) in our context. They denote integers and (possibly higher order) functions, and are built out of numerical and function variables, the constant 0, the successor function s, primitive recursion, abstraction and application.

Definition 3.1 (PA-terms) *(i) PA-terms are all the terms which it is possible to build using the following* term-formation rules.
Let g be a numerical or function variable, G, G_1, G_2 types built out of the type constant Int

using the arrow constructor.

$$g : G \qquad\qquad 0 : Int$$

$$\dfrac{\begin{array}{c}[g : G_1]\\ \vdots \\ p : G_2\end{array}}{\lambda g.p : G_1 \to G_2} \qquad \dfrac{p_1 : G_1 \to G_2 \quad p_2 : G_1}{(p_1 p_2) : G_2}$$

$$\dfrac{u : Int}{\mathsf{s}u : Int} \qquad \dfrac{u : Int \quad p : G \quad f : Int \to G \to G}{\mathsf{Rec}(u, p, f) : G}$$

(ii) On PA-terms the following and well known notions of reductions are defined:

$$\begin{array}{llll} \beta_{PA}) & (\lambda g.p)q & \to_{\beta PA} & p[q/g] \\ \mathsf{Rec}_0) & \mathsf{Rec}(0, p, f) & \to_{Rec_0} & p \\ \mathsf{Rec}_{\mathsf{s}}) & \mathsf{Rec}(\mathsf{s}u, p, f) & \to_{Rec_{\mathsf{s}}} & f(u)\mathsf{Rec}(u, p, f) \end{array}$$

We denote by $\to_{1PA}$ the union of all the above notions of reductions, and by $\to_{PA}$ its reflexive and transitive closure.

(iii) We denote by $\simeq$ the least congruence obtained out of $\to_{PA}$

In what follows, numerical variables (i.e. of type Int) will be denoted by $n, m, \ldots$, while generic PA-terms will be denoted by $u, v, t, \ldots$.

Lemma 3.1 ([Tait 67]) *PA-terms strongly normalizes.*

The types of system $\lambda^{Sym}_{\mathbf{PA}}$ will be like the types of system λ^{Sym}_{Prop}, considering also types corresponding to existential and universal quantification and with the sets of atomic and negated atomic types defined as follows:

$$\mathcal{A} = \{u = v \mid u, v \ PA\text{-terms of type } Int\ \} \quad \mathcal{A}^{\perp} = \{u \neq v \mid u, v \ PA\text{-terms of type } Int\ \}.$$

Definition 3.2 ($\lambda^{Sym}_{\mathbf{PA}}$-rules) - Atomic rules

$$\mathsf{PA}_{id})\ \dfrac{}{\mathsf{PA}_{id} : (n = n)} \qquad \mathsf{PA}_{sym})\ \dfrac{P : (u = t)}{\mathsf{PA}_{sym}(P) : (t = u)}$$

$$\mathsf{PA}_{trans})\ \dfrac{P : (u = v) \quad Q : (v = t)}{\mathsf{PA}_{trans}(P, Q) : (u = t)} \qquad \mathsf{PA}_{bot})\ \dfrac{P : (\mathsf{s}0 = 0)}{\mathsf{PA}_{bot}(P) : \perp}$$

We shall denote by $r)$ a generic atomic rule.

- Logical rules

$$var)\ \frac{}{x^A : A}$$

$$\langle,\rangle)\ \frac{P_1 : A_1 \quad P_2 : A_2}{\langle P_1, P_2\rangle : A_1 \wedge A_2} \qquad \sigma_i)\ \frac{P_i : A_i}{\sigma_i^{A_1,A_2}(P_i) : A_1 \vee A_2}(i = 1,2)$$

$$\lambda)\ \frac{\begin{array}{c}[x : A]\\ \vdots\\ P : \bot\end{array}}{\lambda x.P : A^{\bot}} \qquad \star)\ \frac{P_1 : A^{\bot} \quad P_2 : A}{P_1 \star P_2 : \bot}$$

$$\lambda_\forall)\ \frac{P : A}{\lambda_\forall n.P : \forall n.A}\ (^*) \qquad \sigma_t)\ \frac{P : A(t)}{\sigma_t(P) : \exists n.A(n)}$$

$$\mathsf{Ind})\ \frac{u : Int \quad P : A(0) \quad F : \forall n.A(n) \to A(\mathsf{s}n))}{\mathsf{Ind}(u, P, F) : A(u)}\ (^*_*)$$

$$Conv)\ \frac{P : A(u)}{P : A(u')}\ \text{if}\ \ \mathrm{u} \simeq \mathrm{u}'$$

$(^*)$ *for all $x^B \in FV(P)$ it has to be $n \notin FV(B)$.*
$(^*_*)$ *here and in the following the implication connective is, as said before, a derived one.*

Rules ($\langle,\rangle$), (σ_i), (σ_t) and ($\lambda_\forall$) will be called in the following *introduction* rules.

We shall say that a term P represents the proof of a formula A (is a term of type A) if it is possible to derive $P : A$.

The set of all terms of system $\lambda^{Sym}_{\mathbf{PA}}$ will be denoted by $Terms(\lambda^{Sym}_{\mathbf{PA}})$.

We shall denote by $\Gamma \vdash_{\lambda^{Sym}_{\mathbf{PA}}} P : A$ the fact that $P : A$ is derivable in $\lambda^{Sym}_{\mathbf{PA}}$ from the set of assumptions Γ.

We shall call *atomic* a term formed only by atomic rules.

It is easy to show that all other rules of first order logic natural deduction can be derived from the ones given above.

By PA-closed term we shall mean a term with no free numerical variables in the type of its free variables.

Definition 3.3 ($\lambda^{Sym}_{\mathbf{PA}}$-reduction rules) *We add to the reduction rules of Definition 2.4 the following reductions.*

$$\begin{cases} \beta_\forall) & (\lambda_\forall n.P)\star\sigma_t(Q) \to_{\beta_\forall} P[t/n]\star Q \\ \beta_\forall^\bot) & \sigma_t(Q)\star(\lambda_\forall n.P) \to_{\beta_\forall^\bot} Q\star P[t/n] \end{cases}$$

$$\begin{array}{lll} \mathsf{Ind}_0) & \mathsf{Ind}(0, P, F) & \to_{\mathsf{Ind}_0} P \\ \mathsf{Ind}_\mathsf{S}) & \mathsf{Ind}(\mathsf{s}u, P, F) & \to_{\mathsf{Ind}_\mathsf{S}} F(u)(\mathsf{Ind}(u, P, F))\quad (^*) \end{array}$$

$Comp)\ u \to_{cmp} u'$ *if u and u' are PA-terms and $u \to_{PA} u'$* $(^*_*)$

(*) Here the application of F to u and the application of $F(u)$ to $\mathsf{Ind}(u, P, F)$, corresponding respectively to the rules of elimination of universal quantification and implication, are to be thought as defined in terms of the rules of our system.
($^*_*$) We have this rule in order to be able to reduce PA-terms when they are inside $\lambda^{Sym}_{\mathbf{PA}}$-terms.

The property of strong normalization holds for $\lambda^{Sym}_{\mathbf{PA}}$-terms.

Theorem 3.1 (Strong Normalization for $\lambda^{Sym}_{\mathbf{PA}}$) *Terms of $\lambda^{Sym}_{\mathbf{PA}}$ are strongly normalizable.*

The proof of this theorem can be obtained by an extension of the proof of strong normalization for system λ^{Sym}_{Prop}.

4 Shape of Normal Forms in $\lambda^{Sym}_{\mathbf{PA}}$ and Extraction of Constructive Content

We introduce now two sets of types. From terms whose types are in one of these sets, namely Σ^0_1, it will be possible to extract the constructive content expressed by the types, seen as specifications.

Definition 4.1 *(i) The set of Σ^0_1 types (formulas) is composed by PA-closed m-types not containing base element of the set $\mathcal{A}^\perp$ and no universal quantification, i.e. it is the subset of the PA-closed types of the set defined by the following grammar:*

$$\mathsf{S} ::= \ \mathcal{A} \mid \mathsf{S} \wedge \mathsf{S} \mid \mathsf{S} \vee \mathsf{S} \mid \exists \mathsf{n}.\mathsf{S}$$

where n *ranges over the category of numerical variables.*

(ii) The set of Π^0_1 types (formulas) is composed by the types D such that $D^\perp \in \Sigma^0_1$, i.e. the subset of the PA-closed types of the set defined by the following grammar:

$$\mathsf{P} ::= \mathcal{A}^\perp \mid \mathsf{P} \wedge \mathsf{P} \mid \mathsf{P} \vee \mathsf{P} \mid \forall \mathsf{n}.\mathsf{P}$$

where n *ranges over the category of the numerical variables.*

It is worthwhile to outline that all the results of the previous and the present sections hold also in case we consider any (consistent) set Δ of atomic rules instead of those for Peano Arithmetic. We call λ^{Sym}_{Δ} the calculus obtained out of $\lambda^{Sym}_{\mathbf{PA}}$ replacing its set of atomic rules by a set of atomic rules Δ.

Definition 4.2 *(i) A set Δ of atomic rules is* consistent *if there exists no atomic and closed proof of $\perp$.*

(ii) A system λ^{Sym}_{Δ} is consistent *if there exists no closed proof of $\perp$.*

Lemma 4.1 *The set of atomic rules* **PA** *is consistent.*

We state now the main theorem of this section.

Theorem 4.1 (Main Theorem) *Let P be a closed Σ^0_1-term with type C. Then*

(i) C is atomic $\Rightarrow$ P is atomic.

(ii) C is not atomic $\Rightarrow$ P ends with an introduction (i.e. $(\langle,\rangle)$, (σ_i), (σ_t) or $(\lambda_\forall)$).

The statement of the above theorem is clearly analogous to well known properties of simply typed λ-calculus (it says that the results of a computation over a Σ^0_1-type has a concrete meaning). Its proof however is not at all trivial, since here we are dealing with classical logics.

It is worth remarking that the restriction to Σ^0_1-types in the main theorem is *essential*. For instance, there are closed normal proofs of $(\alpha \vee \alpha^\perp)$ which are not of the form $\sigma_i(P)$, because we can classically prove $(\alpha \vee \alpha^\perp)$ without proving neither α nor $\alpha^\perp$, as the following traditional example shows.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x:\alpha]}{\sigma_1(x):\alpha\vee\alpha^\perp} \quad [y:(\alpha^\perp\wedge\alpha)]}{\sigma_1(x)\star y:\bot}}{\lambda x.(\sigma_1(x)\star y):\alpha^\perp}}{\sigma_2(\lambda x.(\sigma_1(x)\star y)):(\alpha\vee\alpha^\perp)} \quad [y:(\alpha^\perp\wedge\alpha)]}{\dfrac{\sigma_2(\lambda x.(\sigma_1(x)\star y)\star y):\bot}{\lambda y.(\sigma_2(\lambda x.(\sigma_1(x)\star y)\star y):(\alpha\vee\alpha^\perp)}}$$

From the main theorem, to whose proof next subsection will be devoted, it descends easily the consistency of any system λ^{Sym}_Δ, in case the set Δ of atomic rules is consistent.

Corollary 4.1

$$\Delta \textit{ consistent} \Rightarrow \lambda^{Sym}_\Delta \textit{ consistent}$$

It is easy to see that Theorem 4.1 shows an easy and clear way to get what Kreisel obtaind in his [Kre 58], i.e. the extraction of computational content. By Theorem 4.1, from a *normalized* proof of a disjunction we can get a proof of one of the disjuncts, and from a proof of an existentially quantified formula, a witness of it. More in general, given a closed proof of a Σ^0_1 formula in $\lambda^{Sym}_{\mathbf{PA}}$, it is possible to get, by a simple inspection of the normalized proof, the witnesses of all the subformulas of the form $\exists n.A(n)$. This means also that, if we have a formula corresponding to a program specification, i.e. of the form $\forall m.\exists n.A(m,n)$, we can get, out of a proof P of $\forall m.\exists n.A(m,n)$, a recursive function $f : I\!N \to I\!N$ such that for all $k \in I\!N$ $A(k, f(k))$ holds. Fixed $k \in I\!N$, to get $f(k)$ we have simply to normalize the proof $\lambda x^{\forall n.A^\perp(k,n)}.(\sigma_k(x)\star P)$. We outline once more that the reduction rules of our calculus are quite simple and natural, and not ad-hoc devised for constructive content extraction purposes.

4.1 Proof of the Main Theorem

This section will be devoted to the proof of Theorem 4.1. First of all we introduce the class of *minimal* proofs. Our proof will then proceed by first showing, after a series of technical lemmas, that the statement of Theorem 4.1 holds for minimal proofs and then that indeed each closed proof of a sentence in Σ^0_1 is minimal.

Definition 4.3 *A term (proof) of $\lambda^{Sym}_{\mathbf{PA}}$ is* minimal *if it is built out only of rules $(\langle,\rangle)$, (σ_i), (σ_t) and of atomic rules.*

Lemma 4.2 *Let u be a PA-term in normal form. Then either u ends with an introduction (0, s or λ) or it is formed only by eliminations (*Rec *or application) followed by a variable.*

Proof. By induction un u.

- u is a variable.
  Immediate.

- u ends with an introduction rule.
  Immediate.

- u ends with an elimination rule.
  The thesis follows by the induction hypothesis, since if the leftmost immediate subterm of u ended with an introduction rule u would not be in normal form. $\Box$

Lemma 4.3 *(i) Let P be a PA-closed normal term. Then P does not begin with* Ind *(even if* Ind *can occur inside P).*

(ii) Let $P_1 \star P_2$ be a PA-closed term in normal form. Then either P_1 or P_2 is a variable.

Proof. (i) Let us assume, by contradiction, that P is of the form $\mathsf{Ind}(u, Q, F)$. By Lemma 4.2 it follows that either u is of the form $\mathsf{s}v$ or is 0. In both cases we get a contradiction since reduction ($\mathsf{Ind_S}$) or ($\mathsf{Ind_0}$), respectively, could be applied.
(ii) Let us assume, by contradiction, that neither P_1, of type, say, $A^\perp$, nor P_2, of type A, is a variable. First of all we notice that neither P_1 nor P_2 can be of the form $\lambda x.Q$, $\mathsf{Ind}(u, P, F)$ or be applications. The first case is excluded since, otherwise, $P_1 \star P_2$ would not be normal. The second one by (i), while the third case would imply $A \equiv \perp$, which is impossible. Therefore both P_1 and P_2 end either with an introduction or with an atomic rule. We can show that even the latter of these cases is to be escluded since, if one of the two terms ends with an atomic rule, the other one would have a *negated* atomic type and hence could not end neither with an introduction nor with an atomic rule, contradicting what we have just proved. Thus both P_1 and P_2 end with an introduction. It is easy to check that if one ends with $(\langle,\rangle)$ the other has to end with (σ_i) or, alternatively, if one ends with $(\lambda_\forall)$ the other has to end with (σ_t). In both cases, however, we get a contradiction, i.e. $P_1 \star P_2$ would not be normal, being possible to apply reduction $\pi(\pi^\perp)$ or $\beta_\forall(\beta_\forall^\perp)$, respectively. We then conclude that one of P_1, P_2 is necessarily a variable. $\Box$

To continue now toward the complete proof of the Main Theorem we need to introduce one more notion: that of Σ^0_1-term.

Definition 4.4 *Let P be a term and C its type. P is a* Σ^0_1-term *if:*

1. *is PA-closed.*
2. *$C \in \Sigma^0_1$ or $C \equiv \perp$.*
3. *For all $x \in FV(P)$, if D is the type of x then $D \in \Pi^0_1$.*

Lemma 4.4 *Let P be a Σ^0_1-term in normal form, and Q a subterm of its. Then:*
1. Q is a variable x iff it has type in Π^0_1 and
2. Q is not a variable iff it is a Σ^0_1-term.
Moreover, if Q is a variable x then it is on one side of some application $x \star Q'$ or $Q' \star x$ occurring in P.

Proof. By induction over the structure of P.

- $P \equiv x$.
  This case can never occur, since, by definition of Σ^0_1-term, P has to have type in Σ^0_1 and to have the types of its free variables in Π^0_1, and this is impossible.

- $P \equiv \langle P_1, P_2 \rangle$.
  In such a case $P : A_1 \wedge A_2$ with A_1 and A_2 both in Σ^0_1. Moreover the free variables of P_1 and P_2 have types in Π^0_1. Thus P_1 and P_2 are Σ^0_1-terms. Since the strict subterms of P are all the subterms of P_1 and P_2, we get the thesis by the induction hypothesis.

- $P \equiv \sigma_i(P_i)$.
  As in the previous case, we deduce that P_i is a Σ^0_1-term and hence we can obtain the thesis by invoking the induction hypothesis.

- $P \equiv \lambda x.P'$.
  Since $\lambda x.P'$ is a Σ^0_1-term, we get that the type of x is in Π^0_1. Moreover P' has type $\perp$ and $FV(P') \subseteq FV(P) \cup \{x\}$. We then infer that P' is a Σ^0_1-term on which it is possible to apply the induction hypothesis to get the thesis.

- $P \equiv P_1 \star P_2$.
  By Lemma 4.3 (ii), since P is in normal form and Σ^0_1-terms are PA-closed by definition, it follows that P_1 or P_2, say P_1, is a variable x. Then we have that the type of x is in Π^0_1 and therefore P_2 has type in Σ^0_1 and $FV(P_2) \subseteq FV(P)$. This means that P_2 is a Σ^0_1-term. By applying the induction hypothesis on P_2 we get the thesis for P since the strict subterms of P are x or subterms of P_2.

- $P \equiv \lambda_\forall n.P'$.
  This can never be the case since the type of P would be of the form $\forall n.A$ and hence P would not be a Σ^0_1-term.

- $P \equiv \sigma_t(P')$.
  In this case the type of P is of the form $\exists n.A(n)$ with $A(t)$ type of P' and $FV(P') \subseteq FV(P)$. It is easy then to check that P' is a Σ^0_1-term. By applying the induction hypothesis on P' we get the thesis for P since the strict subterms of P are the subterms of P'.

- $P \equiv r(P_1, \ldots, P_n)$. The thesis follows immediately from the induction hypothesis on $P_1, \ldots, P_n$ which have necessarily atomic types and free variables all in $FV(P)$.

- $P \equiv \mathsf{Ind}(u, Q, F)$.
  Such a case can never occur, otherwise we would get a contradiction with Lemma 4.3 (i), since a Σ^0_1-term is PA-closed by definition. □

Corollary 4.2 *Let P be a normal Σ^0_1-term. Then*

(i) P does not contain Ind *symbols.*

(ii) P does not contain $\lambda_\forall$ symbols.

Proof. (i) If there were a term of the form $\mathsf{Ind}(u, P, F)$ then F should have type of the form $\forall n.(A(n)^\perp \vee A(\mathsf{s}n))$ which, by not being Σ^0_1, would lead to a contradiction with Lemma 4.4.
(ii) If there were a term of the form $\lambda_\forall n.P$, it would have type of the form $\forall n.A(n)$ which, by not being Σ^0_1, would lead to a contradiction with Lemma 4.4. □

We can now prove the statement of the main theorem restricted to minimal terms.

Lemma 4.5 *Let P be a minimal, PA-closed, term and let α be $\bot$ or an element in $\mathcal{A}$.*

(i)	$P : \alpha$	$\Rightarrow$	*P is atomic.*
(ii)	$P : A_1 \wedge A_2$	$\Rightarrow$	*P is of the form $\langle P_1, P_2 \rangle$ with P_1 and P_2 minimal.*
(iii)	$P : A_1 \vee A_2$	$\Rightarrow$	*P is of the form $\sigma_i(P_i)$ with P_i minimal.*
(iv)	$P : \exists n.A$	$\Rightarrow$	*P is of the form $\sigma_t(P')$ with P' minimal.*

Proof. (i) By induction on the structure of P. Since P contains only atomic rules and introductions and has an atomic type, it has necessarily the form $r(P_1, \ldots, P_n)$ with r atomic rule. Since the P_i's are minimal, we get the thesis by applying the induction hypothesis to them all.
(ii) (iii) (iv) Since P contains only atomic rules and introductions and has a non-atomic type, we have necessarily that it has the form $\langle P_1, P_2 \rangle$, $\sigma_i(P_i)$, $\sigma_t(P')$, respectively. P_1, P_2, P_i and P' are minimal because P is so. $\square$

As last step we prove now that closed normal Σ^0_1-terms are minimal. The main theorem will then follows by Lemma 4.5.

Lemma 4.6 *Let P be a closed normal Σ^0_1-term.*

(i) If P contains no λ symbol then it is minimal.

(ii) P contains no λ symbol.

Proof. (i) Let P be a closed Σ^0_1-term with no λ symbol in it. By definition, to prove that it is minimal, we have to prove that it does not contain free or bound term variables, symmetric applications and Ind or $\lambda_\forall$ symbols. Since it is closed and variables are bound only by λ's, P cannot contain variables. If P contained an application then, by Lemma 4.3 (ii), it would contain a variable, contradicting what we have just proved. Corollary 4.2 can instead be invoked to make sure that no Ind or $\lambda_\forall$ symbol is present in P.
(ii) We begin by first proving that there exist no subterms of P of the form $\lambda x.Q$ with $x \in FV(Q)$. In order to do that, let us assume, by contradiction that there exist some subterms of the form $\lambda x.Q$ with $x \in FV(Q)$, and let us take a minimal one (w.r.t. the subterm inclusion), say $\lambda x'.Q'$. Since $x' \in FV(Q')$ and P is a normal Σ^0_1-term, by Lemma 4.4 x' occurs necessarily in a subterm $x' \star Q''$ or $Q'' \star x'$. Let us now consider the minimal among such terms, in such a way that $x \notin FV(Q'')$. Thus $Q' \equiv C[x' \star Q'']$ (or $C[Q'' \star x']$) for a context $C[\,]$. Since $\lambda x'.Q'$ is minimal among the subterms with this form and with $x' \in FV(Q')$, we get that $FV(x' \star Q'')$ $(FV(Q'' \star x'))$ $\subseteq FV(Q')$. It follows that $Q' \equiv x \star Q''(Q'' \star x)$, i.e. $C[\,] \equiv [\,]$, otherwise $Q \rightarrow_{Triv} x' \star Q''(Q'' \star x')$, contradicting the hypothesis of normality of P. Even in such a case however we get a contradiction since, by the fact that $x' \notin FV(Q'')$, we could apply a η-($\eta^\perp$-) reduction on $\lambda x'.x' \star Q''$ $(\lambda x'.Q'' \star x')$. Therefore we can infer that there exist no subterms of P of the form $\lambda x.Q$ with $x \in FV(Q)$. From this fact it follows that no variables are discharged in P and hence, since P is closed, all subterms of its are so.
We can now proceed to prove that there exist no subterms of P of the form $\lambda x.Q$ at all. By contradiction, let us assume that there exist subterms $\lambda x'.Q'$ and take a minimal one, in such a way that no λ occurs in Q'. Since $Q' : \bot$, Q' is not a variable and, by Lemma 4.4, it is a closed normal Σ^0_1-term. By point (i)[2] it follows that Q' is a closed minimal proof of $\bot$, and, by Lemma 4.5(i) an atomic one. This contradicts the consistency of the set of atomic rules. $\square$

Lemma 4.7 *Let P be a closed normal Σ^0_1-term. Then P is minimal.*

[2]The need of this result here explains why, in this lemma, the statement of point (i) precedes the one of point (ii).

Proof. Immediate from Lemma 4.6 (ii) and (i). □

We can now give the proof of Theorem 4.1.

Proof of Theorem 4.1. Immediate from Lemmas 4.5 and 4.7.

4.2 A further improvement

From the Lemmas proved above it is possible to derive a further lemma which is quite interesting both from a **theoretical** and **applicative** point of view.

Lemma 4.8 *Let $P(x_1^{N_1}, \ldots, x_k^{N_k})$ be a normal Σ_1^0-term. Then either it is minimal (and hence closed) or it contains a subterm $x_i \star Q_i$ (or $Q_i \star x_i$) where $Q_i^{N_i^\perp}$ is minimal.*

Proof. If P is closed ($k = 0$) apply Lemma 4.7. Otherwise, any x_i has necessarily to occur, by Lemma 4.4, in subterms of the form $x_i \star Q_i$ (or $Q_i \star x_i$). Let us take a minimal one among such terms. We get that, for such minimal term, Q_i is closed and, by Lemma 4.4, is a Σ_1^0-term. Lemma 4.7 enables now us to get the thesis. □

From a **theoretical** point of view the above lemma states that for any $x_1^{N_1}, \ldots, x_k^{N_k} \vdash_{\lambda_\Delta^{Sym}} P : A$ with $A \in \Sigma_1^0$ and $N_i \in \Pi_1^0$, P contains either an example for A or a counterexample for some of N_i. This agrees with the *Curry-Howard* interpretation of P.
P can be seen as a classical proof of

"If $N_1, \ldots N_k$ then P"

and the classical meaning of this is

"Either $N_1^\perp$ or $N_k^\perp$ or P"

From an **applicative** point of view, Lemma 4.8 can be used to speed up the process of extraction of constructive content. If a Σ_1^0-proof $P[R_1/x_1, \ldots, R_k/x_k]$ contains closed terms $R_1, \ldots, R_k$ having Π_1^0-types, *we do not need to consider them during the normalization process*. We can instead replace them by fresh variables, since any normal form of P is indeed a normal form of $P[R_1/x_1, \ldots, R_k/x_k]$[3].
To prove this fact it is enough to observe that if the normal form of P is minimal then it is closed, and hence is is also a normal proof of $P[R_1/x_1, \ldots, R_k/x_k]$. Otherwise, by Lemma 4.8, the normal form of P should have some subterms $x_i \star Q_i$ (or $Q_i \star x_i$) with Q_i minimal. This however leads to a contradiction since, by replacing x_i by R_i, we would get a closed proof of $\perp$, which is impossible.
Informally speaking, as G. Kreisel often said, when we have a proof P of a Π_1^0 type N, we only know that N is inhabited, and nothing else. We can use a fresh variable x^N to build an inhabitant of N, being sure that x will disappear during the normalization process.

[3]We do not know if the converse holds. It would imply that by normalizing $P(x_1, \ldots, x_k)$ we can get all the possible constructive content of $P[R_1/x_1, \ldots, R_k/x_k]$.

5 Strong Normalization for λ^{Sym}_{Prop}

This section will be devoted to the proof of Theorem 2.1. We shall use a non trivial variant of the well known Tait's computability method. We first assign to each type C a set $[\![C]\!]$ (symmetric candidate) of *computable* terms of type C and show that for all $P \in [\![C]\!]$, P strongly normalizes. Then we prove that for each type C, if P has type C then $P \in [\![C]\!]$.

The main property the candidate assignment has to enjoy, in order to make the proof work, consists in the fact that the set of computable terms of a m-type A reflects the way these terms are built. In the present case these properties are the ones stated in the following claim.

Claim 5.1 *There exists an assignment* $[\![-]\!] : A \mapsto [\![A]\!] \subseteq Term_A$ *such that* $[\![\perp]\!] = SN_\perp$ *and*

(i) $Var_A \subseteq [\![A]\!]$.

(ii) $\langle P_1, P_2\rangle \in [\![A_1 \wedge A_2]\!] \quad \Leftrightarrow \quad P_1 \in [\![A_1]\!]$ *and* $P_2 \in [\![A_2]\!]$.

(iii) $\sigma_i(P_i) \in [\![A_1 \vee A_2]\!] \quad \Leftrightarrow \quad P_i \in [\![A_i]\!]$ *(*$i = 1, 2$*).*

(iv) $\lambda x.P \in [\![A]\!] \quad \Leftrightarrow \quad \forall Q \in [\![A^\perp]\!].P[Q/x] \in [\![\perp]\!]$.

For simply typed λ-calculi the properties required for the candidate assignment can be also considered as inductive definition of the sets $[\![A]\!]$. Unfortunately this is not the case for our system. The properties stated above cannot be considered as a definition of $[\![A]\!]$, since clause (iv) would cause a circularity ($[\![A^\perp]\!]$ would be defined in terms of $[\![A]\!]$ which, since $A \equiv A^{\perp\perp}$ would be defined in terms of $[\![A^\perp]\!]$ itself). We have therefore to define candidates with a different method. This will be done in Subsection 5.1 where, besides, properties (i)-(iv) will be proved. Up to then we shall assume Claim 5.1 to hold, i.e. the candidates already well defined for all m-types and properties (i)-(iv) already proved.

Lemma 5.1 *Let A be a type. Then*

$$[\![A]\!] \subseteq SN_A.$$

Proof. By induction on the structure of A, considering the different forms of $P \in [\![A]\!]$.

- $P \equiv x^A$.
  Trivially, $x^A \in SN_A$.

- $P \equiv \langle P_1, P_2\rangle$.
  Then $A \equiv A_1 \wedge A_2$. By Claim 5.1(ii) it follows that $P_i \in [\![A_i]\!]$ ($i = 1, 2$). Now, by the induction hypothesis on $[\![A_i]\!]$ we get $P_i \in SN_{A_i}$. Since each reduction on $\langle P_1, P_2\rangle$ is a reduction on P_1 or P_2, we get $\langle P_1, P_2\rangle \in SN_{A_1\wedge A_2}$.

- $P \equiv \sigma_i(P_i)$.
  Then $A \equiv A_1 \vee A_2$. By Claim 5.1(iii) it follows that $P_i \in [\![A_i]\!]$. Now, by the induction hypothesis and the fact each reduction on $\sigma_i(P_i)$ is indeed a reduction on P_i we get $\sigma_i(P_i) \in SN_{A_1\vee A_2}$.

- $P \equiv \lambda x.P'$.
  Since, by Claim 5.1(i), $x \in [\![A^\perp]\!]$, it follows, by Claim 5.1(iv), that $P' \in SN_\perp$. Each reduction on $\lambda x.P'$ is indeed either a reduction on P' or a $\eta(\eta^\perp)$-reduction and $P' \equiv P'_1\star x (P' \equiv x\star P_1)$. Therefore $\lambda x.P$ has a bound which is at most one more then the bound of P.

- $P \equiv P_1\star P_2$.
  In such a case $P \in [\![\perp]\!] \equiv SN_\perp$. $\square$

Lemma 5.2 *Let $P \in Term_A$. Then*

(i) $\forall Q \in [\![A^\perp]\!].(P \star Q \in SN_\perp) \;\Rightarrow\; P \in [\![A]\!]$.

(ii) $\forall Q \in [\![A^\perp]\!].(Q \star P \in SN_\perp) \;\Rightarrow\; P \in [\![A]\!]$.

Proof. Because of the symmetry of the application, it is enough to prove just one of (i) and (ii), say (i).
Assume $\forall Q \in [\![A^\perp]\!].(P \star Q \in SN_\perp)$. We shall prove $P \in [\![A]\!]$ by induction on the structure of A. We distinguish now different cases according to the form of P. Note that, since A cannot be $\perp$, the case $P \equiv P_1 \star P_2$ cannot occur.

- $P \equiv x^A$.
  Then $P \in [\![A]\!]$ by Claim 5.1(i).

- $P \equiv \langle P_1, P_2 \rangle$.
  Then $A \equiv A_1 \wedge A_2$. By Claim 5.1(ii), it suffices to prove that $P_i \in [\![A_i]\!]$ $(i = 1, 2)$. By the induction hypothesis this can in turn be proved by showing that for all $Q_i \in [\![A_i]\!]$ we have that $P_i \star Q_i \in SN_\perp$. Since $\langle P_1, P_2 \rangle \star \sigma_i(Q_i) \to_1 P_i \star Q_i$, it would be enough to prove $\langle P_1, P_2 \rangle \star \sigma_i(Q_i) \in SN_\perp$. By the assumption $\forall Q \in [\![A^\perp]\!].(P \star Q \in SN_\perp)$, this fact descends from $\sigma_i(Q_i) \in [\![A_1^\perp \vee A_2^\perp]\!]$ which, in turn, is a consequence of Claim 5.1(iii) and of $Q_i \in [\![A_i]\!]$.

- $P \equiv \sigma_i(P_i)$.
  This case can be treated similarly to the previous one.

- $P \equiv \lambda x.P_1$.
  By hypothesis we have that $\forall Q \in [\![A^\perp]\!].((\lambda x.P_1) \star Q \in SN_\perp)$, from which it immediately follows that $\forall Q \in [\![A^\perp]\!].P_1[Q/x] \in SN_\perp$. Hence, by Claim 5.1(iv), we get $\lambda x.P_1 \in [\![A]\!]$. □

Lemma 5.3

$$P \in [\![A]\!],\; P \to_1 P' \;\Rightarrow\; P' \in [\![A]\!].$$

Proof. By induction on A considering the different forms of $P \in [\![A]\!]$.

- $P \equiv x^A$.
  It can never be the case that $x^A \to_1 P'$.

- $P \equiv \langle P_1, P_2 \rangle$.
  In such a case $P \equiv \langle P_1, P_2 \rangle$, $A \equiv A_1 \wedge A_2$ and either $P_1 \to_1 P_1'$ or $P_2 \to_1 P_2'$. In either cases we get the thesis by the induction hypothesis and Claim 5.1(ii).

- $P \equiv \sigma_i(P_i)$.
  This case is similar to the previous one.

- $P \equiv \lambda x.P_1$.
  In such a case we have two possibilities: either $P' \equiv \lambda x.P_1'$ and $P_1 \to_1 P_1'$ or $P_1 \equiv Q_1 \star x (P_1 \equiv x \star Q_1)$ and $P' \equiv Q_1$ (we performed an $\eta(\eta^\perp)$-reduction).

  In the first case, by Claim 5.1(iv), to check that $\lambda x.P_1' \in [\![A]\!]$, it suffices to show, by Claim 5.1, that $\forall Q \in [\![A]\!].P_1'[Q/x] \in SN_\perp$. This fact follows immediately from $P_1'[Q/x] \in SN_\perp$ which in turn follows from the hypothesis and Claim 5.1(iv).

  In the second case, by Lemma 5.2, it suffices to show that $\forall S \in [\![A^\perp]\!].Q_1 \star S \in SN_\perp$ (or equivalently $\forall S \in [\![A^\perp]\!].S \star Q_1 \in SN_\perp$). This follows by the fact $P \equiv \lambda x.Q_1 \star x (\lambda x.x \star Q_1) \in [\![A]\!]$ and Claim 5.1(iv), since $x \notin FV(Q_1)$ and therefore $P_1[S/x] \equiv Q_1 \star S$.

- $P \equiv P_1 \star P_2$.
  Then $P \in [\![\perp]\!] \equiv SN_\perp$ and $P \to_1 P'$ implies $P' \in SN_\perp \equiv [\![\perp]\!]$. $\square$

Lemma 5.4

$$P \in [\![A]\!], Q \in [\![A^\perp]\!] \;\; \Rightarrow \;\; P \star Q \in SN_\perp$$

Proof. By Lemma 5.1, there exist n and m bounds for P and Q, respectively. We prove $P \star Q \in SN_\perp$ by double induction: primary induction on the structure of A and secondary induction on $n+m$.
It is easy to see that proving the thesis is equivalent to proving that

$$\forall S.(P \star Q \to_1 S \;\; \Rightarrow \;\; S \in SN_\perp)$$

We have eight cases, pairwise symmetric, to consider.

1. $\begin{cases} P \equiv \lambda x.P_1, S \equiv P_1[Q/x] \\ Q \equiv \lambda x.Q_1, S \equiv Q_1[P/x] \end{cases}$

2. $\begin{cases} P \to_1 P', S \equiv P' \star Q \\ Q \to_1 Q', S \equiv P \star Q' \end{cases}$

3. $\begin{cases} P \star Q \to_{Triv} S, \;\; S \text{ subterm of } P \\ P \star Q \to_{Triv} S, \;\; S \text{ subterm of } Q \end{cases}$

4. $\begin{cases} P \equiv \langle P_1, P_2 \rangle, Q \equiv \sigma_i(Q_i), S \equiv P_i \star Q_i \\ P \equiv \sigma_i(P_i), Q \equiv \langle Q_1, Q_2 \rangle, S \equiv P_i \star Q_i \end{cases}$

1. For these cases the thesis follows immediately from Claim 5.1(iv).
2. By Lemma 5.3 we have that $P' \in [\![A]\!]$ or $Q' \in [\![A]\!]$. We then get $P' \star Q \in SN_\perp$ or $P \star Q' \in SN_\perp$ by the induction hypothesis, since in both cases the sum of the two bounds decreased.
3. Since S is a subterm of P or Q, the thesis follows immediately from the hypothesis and Lemma 5.1.
4. Immediate by primary induction. $\square$

We now need a last lemma in order to prove strong normalization.

Lemma 5.5 *Let C be any type and $P \in Term_C$ with $FV(P) \subseteq \{x_1^{A_1}, \ldots, x_n^{A_n}\}$. Then*

$$\forall P_1 \in [\![A_1]\!] \ldots P_n \in [\![A_n]\!].P[P_1/x_1, \ldots, P_n/x_n] \in [\![C]\!]$$

Proof. By induction on P.

- $P \equiv x$
  In such a case $P \equiv x_i$ and $C \equiv A_i$ for some i by hypothesis.
  Hence $P[P_1/x_1, \ldots, P_n/x_n] \equiv P_i \in [\![A_i]\!] \equiv [\![C]\!]$
- $P \equiv \langle P_1, P_2 \rangle, \sigma_i(P_i), P_1 \star P_2$
  In all these cases the thesis follows by the induction hypothesis on P_1, P_2 and P_i, and Claim 5.1(ii), 5.1(iii) and 5.4, respectively.

- $\lambda x.P'$.
  Since we can rename bound variables, it is not restrictive to assume $x \notin \{x_1^{A_1}, \ldots, x_n^{A_n}\}$. Now, by Claim 5.1(iv), to prove that

$$(\lambda x.P')[P_1/x_1, \ldots, P_n/x_n] \equiv \lambda x.P'[P_1/x_1, \ldots, P_n/x_n] \in [\![C]\!]$$

  it is enough to prove that for all $Q \in [\![A']\!]$, with A' type of x, $P'[Q/x, P_1/x_1, \ldots, P_n/x_n] \in [\![C]\!]$. This last fact follows by the induction hypothesis. □

We can now present the proof of Theorem 2.1.

Proof of Theorem 2.1. One inclusion is immediate by definition. For the other one, let $P \in Term_C$ with $FV(P) = \{x_1^{A_1}, \ldots, x_n^{A_n}\}$. By Claim 5.1(i) we get $x_i \in [\![A_i]\!]$ $(i = 1, \ldots, n)$ and therefore, by Lemma 5.5 $P \equiv P[x_1/x_1, \ldots, x_n/x_n] \in [\![C]\!]$. Lemma 5.1 allows now to infer $P \in SN_C$. □

5.1 Symmetric Candidates: Definition and Main Properties

This subsection will be devoted to the definition of the sets of computable terms, which we shall call *symmetric candidates*. We have spoken before about the impossibility of considering the properties of Claim 5.1 as an inductive definition for the symmetric candidates.

We first define an operator on sets of terms of a given type for each term constructor but the symmetric application ($Pair(-)$, $Sigma(-)$, $Lambda(-)$). Out of these operators we then define an operator for the negation ($Neg(-)$), reflecting the possible way of obtaining terms whose type can be seen as a negation. Since we wish the involutive property of negation to be reflected in the symmetric candidates, we define a candidate as a fix point of the composition of Neg with itself. Since these composition turns out to be an increasing operator, the fix point exists by Tarsky Theorem. The definition of Neg use the notion of symmetric candidate. This is why the definitions of Neg and of symmetric candidate will have to be given simultaneously on the structure of the type.

Definition 5.1 *Let A, A_1, A_2 be types. We define the operators*

$$\begin{array}{ll} Pair_{A_1,A_2}: & \mathcal{P}(Term_{A_1}) \times \mathcal{P}(Term_{A_2}) \to \mathcal{P}(Term_{A_1 \wedge A_2}) \\ Sigma^i_{A_1,A_2}: & \mathcal{P}(Term_{A_i}) \to \mathcal{P}(Term_{A_1 \vee A_2}) \\ Lambda_A: & \mathcal{P}(Term_{A^\perp}) \to \mathcal{P}(Term_A) \end{array}$$

as follows.

$$\begin{array}{lll} Pair_{A_1,A_2}(X_1, X_2) & =_{Def} & \{\langle P_1, P_2\rangle \in Term_{A_1 \wedge A_2} \mid P_1 \in Term_{A_1} \textit{ and } P_2 \in Term_{A_2}\}. \\ Sigma^i_{A_1,A_2}(X_i) & =_{Def} & \{\sigma_i(P_i) \in Term_{A_1 \vee A_2} \mid P_i \in X_i\}\ (i = 1, 2) \\ Lambda_A(X) & =_{Def} & \{\lambda x.P \in Term_A \mid \forall Q \in X.P[Q/x] \in SN_\perp\} \end{array}$$

Definition 5.2 *Let A be a m-type. By induction on the structure of A we simultaneously define*

(a) The operators:

$$\begin{cases} Neg_A : \mathcal{P}(Terms_{A^\perp}) \to \mathcal{P}(Terms_A) \\ Neg_{A^\perp} : \mathcal{P}(Terms_A) \to \mathcal{P}(Terms_{A^\perp}) \end{cases}$$

(b) The sets $[\![A]\!]$ and $[\![A^\perp]\!]$.

as follows.

(a) Let $X \subseteq Term_A$ and $Y \subseteq Term_{A^\perp}$. By the involutive property of negation in our system it is not restrictive to consider only the cases A atomic and A conjunction.

- $A \equiv \alpha$.

$$\begin{cases} Neg_\alpha(Y) =_{Def} Var_\alpha \cup Lambda_\alpha(Y) \\ Neg_{\alpha^\perp} =_{Def} Var_{\alpha^\perp} \cup Lambda_{\alpha^\perp}(X) \end{cases}$$

- $A \equiv A_1 \wedge A_2$ $(A^\perp \equiv A_1^\perp \vee A_2^\perp)$.

$$\begin{cases} Neg_{A_1 \wedge A_2}(Y) =_{Def} Var_{A_1 \wedge A_2} \cup Pair([\![A_1]\!], [\![A_2]\!]) \cup Lambda_{A_1 \wedge A_2}(Y) \\ Neg_{A_1^\perp \vee A_2^\perp}(X) =_{Def} Var_{A_1^\perp \vee A_2^\perp} \cup (\bigcup_{i=1}^2 Sigma^i([\![A_i]\!])) \cup Lambda_{A_1^\perp \vee A_2^\perp}(X). \end{cases}$$

(b) For all A, Neg_A is a decreasing operator (w.r.t. set theoretical inclusion), since $Lambda_A$ is so. Then, once one has defined Neg_A for some A, it is possible to get, by Tarsky's Fixed Point Theorem, the smallest fixpoint of the (increasing) operator $Neg_A \circ Neg_{A^\perp}$. Let us call it X_0. We then define:

$$\begin{cases} [\![A^\perp]\!] =_{Def} X_0 \\ [\![A^\perp]\!] =_{Def} Neg_{A^\perp}(X_0) \end{cases}$$

We extend the definition of computable set of terms to all types by defining $[\![\perp]\!] =_{Def} SN_\perp$.

Note that, for our strong normalization proof, the use of the *smallest* fixpoint in the above definition is irrilevant.

Since $[\![A]\!]$ is a fixed point of $Neg_A \circ Neg_{A^\perp}$ we get what we where looking for, i.e. an operator with the property $[\![A]\!] \equiv Neg_A([\![A^\perp]\!])$.

Given a m-type A, from the fact $[\![A]\!] \equiv Neg_A([\![A^\perp]\!])$ and the definition of Neg_A, the properties $(i) - (iv)$ of Claim 5.1 descend easily.

References

[BB 91] F. Barbanera, S. Berardi, "Witness Extraction in Classical Logic through Normalization", to appear in *Proceedings of BRA-LF workshop*, Cambridge University Press.

[BB 92] F. Barbanera, S. Berardi, "A constructive valuation interpretation for classical logic and its use in witness extraction", *Proceedings of Colloquium on Trees in Algebra and Programming (CAAP)*, LNCS 581, Springer Verlag, 1992.

[BB 93] F. Barbanera, S. Berardi, "Extracting constructive content from classical proofs via control like reductions", *Proceedings of the International conference on Typed Lambda Calculus and applications (TLCA) 93*, LNCS, Springer Verlag, 1993..

[Con 86] R.L. Constable, S. Allen, H. Bromely, W. Cleaveland et al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[Coq 92] T. Coquand, "A Game-theoric semantic of Classical Logic", 1992, submitted to *JSL*.

[Fri 78] H. Friedman, "Classically and intuitionistically provably recursive functions", in *Higher Set Theory*, eds. Scott D.S. and Muller G.H., Lecture Notes in Mathematics vol.699, 21-28, Springer Verlag, 1978.

[Gir 72] J.-Y. Girard, *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre superiéur*, Thése de Doctorat d'Etat, University of Paris, 1972.

[Gri 90] T.G. Griffin, "A formulae-as-types notion of control", in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming languages*, 1990.

[Kre 58] G. Kreisel, "Mathematical significance of consistency proofs", *Journal of Symbolic Logic*, 23:155-182, 1958.

[Mur 90] C. Murthy, *Extracting constructive content from classical proofs*, Ph.D. thesis, 90-1151, department of Computer Science, Cornell University, 1990.

[NPS 90] B. Nordström, K. Petersson and J. Smith, *Programming in Martin-Löf's Type Theory*, OUP, 1990.

[Par 92] M. Parigot, $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction, *Proceedings LPAR'92*, Lecture Notes in Computer Science vol. 624, 190-201, 1992.

[PN 90] L.C. Paulson and T. Nipkow, "Isabelle tutorial and user's manual", technical report 189, University of Cambridge, 1990.

[Pra 65] D. Prawitz, *Natural deduction, a proof theoretical study*, Stockolm, Almqvist and Winskell, 1965.

[Pra 81] D. Prawitz, "Validity and normalizability of proofs in 1-st and 2-nd order classical and intuitionistic logic", in *Atti del I Congresso Italiano di Logica*, Napoli, Bibliopolis, 11-36, 1981.

[Tait 67] W.W. Tait, "Intensional interpretation of functional of finite types", *journal of Symbolic Logic* 32, 1967.

Towards Checking Proof-Checkers

Robert S. Boyer [§]
University of Texas, Austin TX 78712-1188, USA,
Gilles Dowek,[¶]
INRIA, B.P. 105, 78153 Le Chesnay Cedex, France

It is a common practice in software engineering to use one's own tools. For instance, compilers for programming languages are usually written in a compiled programming language. The past thirty years has seen the construction of many *proof development systems*, which in some cases permit one to develop a program together with a proof of its correctness with respect to some specification. The methodology of developing proved programs is intended to become a standard way of programming computers, so proof development systems themselves ought ultimately to be developed using these techniques. A novel issue that arises with proving the correctness of proof development systems is *cross-verification*, i.e., proving the correctness of one system with another. In this way, if one trusts one system, one may trust the other. In this paper we describe an implementation of the Calculus of Constructions [2, 10] in the logic of Nqthm, also known as the Boyer-Moore system [5], and we describe an initial foray into proving theorems about this implementation. Our work is complementary to the deep work described in [3] [2] [13] as our goal is not to prove meta-theoretical properties of the Calculus of Constructions (normalization, confluence, etc.), but rather use these meta-theoretical properties to prove the correctness of an actual implementation of a type-checker, one that can run under any LISP interpretor.

1 Double Checkers and Proof-objects

Less complex by far than a whole proof development system is a *proof-checker*, a program that takes as an input a proposition P and a proof p and checks that the proof p is indeed a proof of P. Formal proofs are typically tedious to write, so proof development systems are often much more complex than mere proof-checkers, and are thus much more difficult to specify and prove correct. Nevertheless, the correctness of a proof is independent of the proof development process used to build it and once obtained such a proof can always be re-checked by a mere proof-checker. Thus, in order to construct a safe proof development system, we do not need to prove the correctness of the entire proof development system but merely that of a proof-checker for the same logic.

[§] boyer@cli.com
[¶] Gilles.Dowek@inria.fr

We get the following schema:

To following such a schema, we need to have a well-defined proof-language, which is used as output by the proof development system and as input by the proof-checker. For these purposes, languages based on Heyting semantics and the Curry-Howard isomorphism, such as the Calculus of Constructions, are good candidates.

As we will not necessarily check the correctness of the whole proof development system, but only the proof-checker, we have to make sure that the re-checked proof is indeed a proof of the proposition P. So we have to be able to read the statement (as well as the axioms, definitions, etc.) produced by the proof development system and used by the proof-checker. We need therefore to write a pretty-printer for the intermediate language.

In the investigations discussed in this paper, we have used the system COQ [11] as our proof development system, and we have specified and implemented a proof-checker for the Calculus of Constructions (the logic of COQ) in Nqthm.

2 The Calculus of Constructions

2.1 Basic Formalism

The basic idea underlying systems based on Heyting semantics and the Curry-Howard isomorphism (such as the Calculus of Constructions) is that a proof of a proposition is a functional object. For instance a proof of a proposition of the form $A \Rightarrow B$ is a function that maps every proof of A to a proof of B. The type of this function is isomorphic to the proved proposition, so types and proposition are identified, as are objects and proofs. In order to express all the proofs, the term language has to be an extension of typed lambda-calculus supporting functions from terms to types (dependent types), functions from types to terms (polymorphic types) and functions from types to types (type constructors). A smooth presentation is obtained when we take only one syntactical category for terms and types.

The basic judgement in this formalism is written $\Gamma \vdash t : T$ which is read t *has type* T *in the context* Γ. The context Γ contains the types of the free variables of t and T. An additional judgement, written Γ *is well-formed*, indicates that the context Γ is valid, i.e., the type of each variable x declared in Γ is a term well-typed in the part of the context declared in the left of x.

This calculus is defined in the following way.

Definition: Term

The set of terms is the smallest set such that:

- $Prop$ is a term,
- $Kind$ is a term,
- if x is an identifier then x is a term,

- if t and t' are terms then $(t\ t')$ is a term,
- if t and t' are terms and x is an identifier then $[x : t]t'$ is a term,
- if t and t' are terms and x is an identifier then $(x : t)t'$ is a term.

The terms $Prop$ and $Kind$ are two predefined types, $Prop$ is the type of the types (and of the propositions) and $Kind$ is the type of the term $Prop$. The terms $(t\ t')$ are *applications*, the terms $[x : t]t'$ are λ-*abstractions*, and the terms $(x : t)t'$ are *products*. The product $(x : t)t'$ is a generalization of the type $t \rightarrow t'$. The notation $t \rightarrow t'$ is used for $(x : t)t'$ when x does not occur free in t'.

At first in this presentation, we shall ignore variable renaming problems. We subsequently treat this matter precisely, after we introduce the de Bruijn notation for bound variables.

Definition: Context

A *context* Γ is a list of pairs (written $x : T$) where x is an identifier and T a term, i.e., the set of contexts is the smallest set such that:

- $[\,]$ is a context,
- if Γ is a context, x an identifier and T a term then $\Gamma[x : T]$ is a context.

The term T is called the type of the variable x.

Definition: Substitution

Let t and u be terms and x an identifier. The term $t[x \leftarrow u]$ is defined by induction over the structure of t as:

- $Prop[x \leftarrow u] = Prop$,
- $Kind[x \leftarrow u] = Kind$,
- $x[x \leftarrow u] = u$,
- $y[x \leftarrow u] = y$, when y is an identifier different from x,
- $(t\ t')[x \leftarrow u] = (t[x \leftarrow u]\ t'[x \leftarrow u])$,
- $([y : t]\ t')[x \leftarrow u] = ([y : t[x \leftarrow u]]t'[x \leftarrow u])$,
- $((y : t)\ t')[x \leftarrow u] = ((y : t[x \leftarrow u])t'[x \leftarrow u])$.

Note that the definition of the substitution function is effective. This function is simple enough that no non-effective definition seems more intuitive than this one. Let us reiterate that we are ignoring variable renaming problems at this point.

Definition: β-reduction

The β-reduction relation $\triangleright$ is the smallest relation such that $([x : T]t\ u) \triangleright t[x \leftarrow u]$ and that is reflexive and transitive and is also a congruence with respect to term structure, i.e., the smallest relation such that:

- $([x : T]t\ u) \triangleright t[x \leftarrow u]$,

- $t \rhd t$,
- if $t \rhd u$ and $u \rhd v$ then $t \rhd v$,
- if $t \rhd u$ and $t' \rhd u'$ then $(t\ t') \rhd (u\ u')$,
- if $t \rhd u$ and $t' \rhd u'$ then $[x:t]t' \rhd [x:u]u'$,
- if $t \rhd u$ and $t' \rhd u'$ then $(x:t)t' \rhd (x:u)u'$.

Definition: β-convertibility

The relation $\equiv$ is the smallest equivalence relation that contains β-reduction, i.e., the smallest relation such that:

- if $t \rhd u$ then $t \equiv u$,
- if $t \equiv u$ then $u \equiv t$,
- if $t \equiv u$ and $u \equiv v$ then $t \equiv v$.

Definition: Typing rules

The relations Γ *is well-formed* and t *has type* T *in* Γ (for which we use the notation $\Gamma \vdash t : T$) are the smallest relations such that:

- $[\,]$ is well-formed,
- if $\Gamma \vdash T : s$ and $s \in \{Prop, Kind\}$ then $\Gamma[x:T]$ is well-formed,
- if Γ is well-formed then $\Gamma \vdash Prop : Kind$,
- if Γ is well-formed and $x : T \in \Gamma$ then $\Gamma \vdash x : T$,
- if $\Gamma \vdash T : s$, $\Gamma[x:T] \vdash U : s'$, $s \in \{Prop, Kind\}$ and $s' \in \{Prop, Kind\}$ then $\Gamma \vdash (x : T)U : s'$,
- if $\Gamma \vdash T : s1$, $\Gamma[x:T] \vdash U : s2$, $\Gamma[x:T] \vdash t : U$, $s1 \in \{Prop, Kind\}$ and $s2 \in \{Prop, Kind\}$ then $\Gamma \vdash [x:T]t : (x:T)U$,
- if $\Gamma \vdash t : (x:T)U$ and $\Gamma \vdash u : T$ then $\Gamma \vdash (t\ u) : U[x \leftarrow u]$,
- if $\Gamma \vdash t : T$, $\Gamma \vdash U : s$ and $T \equiv U$ and $s \in \{Prop, Kind\}$ then $\Gamma \vdash t : U$.

2.2 Proof-checker

A *proof-checker* is a program that takes a context Γ, and two terms t and T and gives back the boolean value *true* if $\Gamma \vdash t : T$ and *false* otherwise. The claim that a proof-checker exists, is constructively speaking, the claim that the ternary predicate $\Gamma \vdash t : T$ is decidable.

2.3 Example

Let us consider a type T, three elements of type T: a, b, and c, a relation R over the elements of type T, an axiom $trans$ stating that the relation R is transitive, and two axioms $ax1$ (resp. $ax2$) stating that the elements a and b (resp. b and c) are related by R, i.e., the context
$\Gamma = [T : Prop; a : T; b : T; c : T; R : T \to T \to Prop;$
$trans : (x : T)(y : T)(z : T)((R\ x\ y) \to (R\ y\ z) \to (R\ x\ z)); ax1 : (R\ a\ b); ax2 : (R\ b\ c)]$
In this context the term $(trans\ a\ b\ c\ ax1\ ax2)$ has type $(R\ a\ c)$ (i.e., the term $(trans\ a\ b\ c\ ax1\ ax2)$ is a proof of the proposition $(R\ a\ c)$).

2.4 Definitions and Lemmas

In mathematical developments, *definitions* are useful; they permit us to associate a name x with a term t of type T and then to use x in place of t. When the term t is seen as a proof, its type, the definition $x := t : T$, is called a *lemma*.

The Calculus of Constructions can be enhanced with definitions and lemmas in the following way.

In the definition of the notion of *context* we add the rule:

- if Γ is a context, x an identifier and t and T terms then $\Gamma[x := t : T]$ is a context.

β-reduction and β-convertibility are replaced by $\beta\delta$-reduction and $\beta\delta$-convertibility. These relations are now parameterized by a context.

Definition: $\beta\delta$-reduction
The $\beta\delta$-reduction relation $\rhd_\Gamma$ is the smallest relation such that:

- $([x : T]t\ u) \rhd_\Gamma t[x \leftarrow u]$,
- if $x := t : T \in \Gamma$ then $x \rhd_\Gamma t$,
- $t \rhd_\Gamma t$,
- if $t \rhd_\Gamma u$ and $u \rhd_\Gamma v$ then $t \rhd_\Gamma v$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $(t\ t') \rhd_\Gamma (u\ u')$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $[x : t]t' \rhd_\Gamma [x : u]u'$,
- if $t \rhd_\Gamma u$ and $t' \rhd_\Gamma u'$ then $(x : t)t' \rhd_\Gamma (x : u)u'$.

Definition: $\beta\delta$-convertibility
The $\beta\delta$-convertibility relation $\equiv_\Gamma$ is the smallest equivalence relation that contains $\beta\delta$-reduction, i.e., the smallest relation such that:

- if $t \rhd_\Gamma u$ then $t \equiv_\Gamma u$,
- if $t \equiv_\Gamma u$ then $u \equiv_\Gamma t$,
- if $t \equiv_\Gamma u$ and $u \equiv_\Gamma v$ then $t \equiv_\Gamma v$.

Definition: Typing rules

Finally, we modify the last typing rule in:

- if $\Gamma \vdash t : T$, $\Gamma \vdash U : s$, $T \equiv_\Gamma U$ and $s \in \{Prop, Kind\}$ then $\Gamma \vdash t : U$,

and we add two typing rules

- if $\Gamma \vdash t : T$ then $\Gamma[x := t : T]$ is well-formed,
- if Γ is well-formed and $x := t : T \in \Gamma$ then $\Gamma \vdash x : T$.

In this enhanced formalism, we can add to the context Γ above the lemma $(R\ a\ c)$ and get the well-formed context $\Gamma[lem := (trans\ a\ b\ c\ ax1\ ax2) : (R\ a\ c)]$.

In this formalism a proof-checker can be merely a program that takes a context Γ as input and answers whether this context is well-formed or not; indeed, we have $\Gamma \vdash t : T$ if and only if the context $\Gamma[x := t : T]$ is well-formed.

3 A Cursory Overview of the Logic of Nqthm

For a complete description of the Nqthm logic, we refer the reader to Chapter 4 of [5]. We now make a few vague remarks that we hope will permit those unfamiliar with this logic to read the subsequent formulas written in the logic. The logic of Nqthm is a quantifier-free first order logic with equality. The syntax is Lisp-like. The basic theory includes axioms defining the following:

- the Boolean constants `t` and `f`, corresponding to the true and false truth values.
- equality. `(equal x y)` is `t` or `f` according to whether `x` is equal to `y`.
- an if-then-else function. `(if x y z)` is `z` if `x` is `f` and `y` otherwise.

The logic of Nqthm contains two 'extension' principles under which the user can introduce new concepts into the logic with the guarantee of consistency.

- *The Shell Principle* allows the user to add axioms introducing 'new' inductively defined 'abstract data types.' Natural numbers, ordered pairs, and symbols are axiomatized in the logic by adding shells:
  - *Natural Numbers.* The nonnegative integers are built from the constant `0` by successive applications of the constructor function `add1`. The function `numberp` recognizes natural numbers. The function `sub1` returns the predecessor of a non-`0` natural number.
  - *Symbols.* The data type of symbols, e.g., `'kind`, is built using the primitive constructor `pack` and `0`-terminated lists of ASCII codes. The symbol `'nil`, also abbreviated `nil`, is used to represent the empty list.
  - *Ordered Pairs.* Given two arbitrary objects, the function `cons` builds an ordered pair of these two objects. The function `listp` recognizes ordered pairs. The functions `car` and `cdr` return the first and second component of such an ordered pair. Lists of arbitrary length are constructed with nested pairs. Thus (`list` $arg_1\ \ldots\ arg_n$) is an abbreviation for (`cons` arg_1 ... (`cons` arg_n `nil`)).

- *The Definitional Principle* allows the user to define new functions in the logic. For recursive functions, there must be an ordinal measure of the arguments that can be proved to decrease in each recursion, which, intuitively, guarantees that one and only one function satisfies the definition. Many functions are added as part of the basic theory by this definitional principle.

The rules of inference of the logic are those of propositional logic and equality with the addition of mathematical induction.

Commands to the theorem prover include

- `(dcl` fn `(x y))`, which declares fn to be an undefined function of two arguments.
- `(defn` fn `(x y)` body`)`, which defines the function fn to take two arguments, x and y, and to return body as the value of (fn x y).
- `(add-axiom` name `(types ...)` formula`)`, which adds formula as an axiom, storing it under name and suggesting how best to use the formula in proofs with the hints types ....
- `(prove-lemma` name `(types ...)` formula`)`, which adds formula as a proved lemma after proving it, storing it under name and suggesting how best to use the formula in proofs with the hints types ....

4 Inductive Definitions

4.1 Inductive Definitions

Most of the definitions of section 2 are *inductive definitions*, i.e., definitions of sets (or predicates) of the following form:
A is the smallest subset of B such that, if $x_1, ..., x_{n_1}$ are element of A then $f_1(x_1, ..., x_{n_1})$ is an element of A, ... and if $x_1, ..., x_{n_p}$ are elements of A then $f_p(x_1, ..., x_{n_p})$ is an element of A.
The existence of such a set is given by Tarski's fixed-point theorem, by considering the function from $\mathcal{P}(B)$ to $\mathcal{P}(B)$ that associates to the set X the set

$$F(X) = \{f_i(a_1, ..., a_{n_i}) \mid 1 \leq i \leq p \wedge a_1, ..., a_{n_i} \in X\}.$$

This function is obviously increasing for the order $\subset$ in $\mathcal{P}(B)$. The set A is defined as its least fixed point. This least fixed-point is the set

$$A = \bigcap_{X \in C} X$$

Where $C = \{X \in \mathcal{P}(B) \mid \forall 1 \leq j \leq p \; \forall x_1, ..., x_{n_j} \in X(f_j(x_1, ..., x_{n_j}) \in X)\}$.
It is also the case that

$$A = \bigcup_{i \in \mathcal{N}} F^i(\emptyset)$$

Finally, an element a is in A if and only if there exists a sequence $a_1, ..., a_k$ such that $a_k = a$ and for each i there exists an f_j and elements $b_1, ..., b_{n_j}$ of $\{a_1, ..., a_{i-1}\}$ such that $a_i = f_j(b_1, ..., b_{n_j})$ [2].

Even if the functions $f_1, ..., f_p$ are recursive, the defined set A may not be recursive (e.g., the set of theorems of arithmetic can be inductively defined, although it is not recursive), but it is recursively enumerable.

4.2 Inductive Definitions as Specifications

The inductive definition given above is a quite natural specification for the predicate *well-formed*. But, of course, since inductive definitions are not effective, it is not an implementation. (More generally, defining a predicate using quantification over an infinite domain may lead to a clear specification but not necessarily to an obvious implementation.) We want to give an effective definition of a predicate *check* (as a LISP program) and prove that *check* implements the predicate *well-formed*, i.e., that these two predicates are extentionally equivalent. More precisely we wish to prove the *soundness* of the implementation:

$$\forall \Gamma\ ((\mathit{check}\ \Gamma) \Rightarrow (\mathit{well\text{-}formed}\ \Gamma))$$

and its *completeness*:

$$\forall \Gamma\ ((\mathit{well\text{-}formed}\ \Gamma) \Rightarrow (\mathit{check}\ \Gamma))$$

4.3 An example of a Specification and a Program

Before we continue with the implementation of the Calculus of Constructions, let us give as an illustration a tiny example of a predicate, defined both as an inductive predicate and as a program.

Definition The predicate *even* is the smallest predicate that contains 0 and $n+2$ if it contains n.

Definition The predicate `ev` is defined by the following algorithm in the Lisp-like logic of Nqthm:

```
(defn ev (x)
 (if (numberp x)
     (if (equal x 0) t (if (equal x 1) f (ev (sub1 (sub1 x)))))
     f))
```

The *soundness* of the implementation is:

$$\forall x\ ((ev\ x) \Rightarrow (even\ x))$$

and the *completeness* is:

$$\forall x\ ((even\ x) \Rightarrow (ev\ x))$$

4.4 Inductive Definitions in a First Order Setting

Inductive definitions in the foregoing style (the least set such that ...) are typically expressed either within a first order logic containing axioms for set theory or in a high-order logic. One might be attempted to express the inductive definition of *even* in the first-order logic of Nqthm thus:

```
(add-axiom even0 () (even 0))
(add-axiom even_plus_two () (implies (even x) (even (add1 (add1 x)))))
```

But these axioms only express that the set of even numbers contains 0 and $n+2$ if it contains n (positive part of the definition), but not the fact that it is the smallest set verifying these properties (negative part). So, for instance the statement *(not (even 1))* is not provable from these axioms. Roughly speaking the positive part of an inductive definition (the fact that the defined set verifies the given properties) is useful to prove the soundness of an implementation and the negative part (the fact that it is the smallest among these sets) is useful to prove its completeness. For instance with the two axioms above we can prove the soundness of the implementation:

$$\forall x \ ((ev\ x) \Rightarrow (even\ x))$$

but not its completeness:

$$\forall x \ ((even\ x) \Rightarrow (ev\ x))$$

An approach to handling completeness in the Nqthm logic is to define the predicate `ev` as above and then prove that it verifies the two conditions:

```
(prove-lemma corr ()
     (and (ev 0)
          (implies (ev x) (ev (add1 (add1 x))))))
```

and then that every predicate `fn` that verifies these two properties contains the predicate `ev`. We first declare `fn` to be an undefined function of one argument:

```
(dcl fn (x))

(add-axiom fn-prop ()
   (and (fn 0) (implies (fn x) (fn (add1 (add1 x))))))

(prove-lemma comp () (implies (ev x) (fn x)))
```

The second order quantification *every predicate* `fn` is simulated in this first order setting by extending the language with a new predicate symbol `fn`. Although it is very powerful, this technique sometimes leads to surprisingly long and uncomfortable proofs.

Another solution is to add as axioms more properties of the specified predicate. For instance, for the definition of `even` one can add the axioms:

```
(add-axiom even1 () (not (even 1)))
(add-axiom even_minus_two () (implies (even (add1 (add1 x))) (even x)))
```

Alternatively one can add the axiom:

```
(add-axiom even_inv () (implies (even x) (or (equal x 0)
                                              (and (leq 2 x)
                                                   (even (sub1 (sub1 x)))))))
```

The generality of this approach (i.e., the possibility of expressing the negative part of an inductive definition by a first order statement) is not yet understood by us [1].

In this paper we have only proved the soundness of our implementation (and not its completeness), so we have merely stated as axioms the positive part of the inductive definitions. Nevertheless, although we have only worked on the soundness half, we have failed to prove three lemmas presented below, which are useful in the soundness proof but which require the negative part of the inductive definition.

5 Normalization

The kernel of the implementation of our proof-checker is the normalization function that permits us to decide if two terms are convertible. In the recursive fragment of the Nqthm logic, only total functions can be defined, so one would need to prove the normalization of reduction to define it. The following problems arise: (1) the reduction function does not terminate on all terms, but only on well-typed terms and it is not possible in the Nqthm logic to define a function restricted in such a way as to be applicable only to some terms, (2) even on typed terms, the ordinal of the normalization function is far above ε_0 and therefore this function cannot be defined in the recursive fragment of the Nqthm logic. It is possible to define any partial recursive function in the Nqthm logic via the function `EVAL$`, an interpreter for the partial recursive functions. However, reasoning about functions defined via `EVAL$` within Nqthm is much more difficult than reasoning about recursively defined functions.

The current solution we have taken in our implementation is to give a bound c to the normalization function, such that it stops after c reduction steps. This way, we get weaker soundness and completeness statements.

Soundness:

$$\forall \Gamma \ (\exists c \ (\textit{check} \ \Gamma \ c) \Rightarrow (\textit{well-formed} \ \Gamma))$$

Completeness:

$$\forall \Gamma \ ((\textit{well-formed} \ \Gamma) \Rightarrow (\exists c \ \textit{check} \ \Gamma \ c))$$

Such a parameter as c above, sometimes called a 'clock' parameter, is frequently employed by users of Nqthm in the verification of computing systems, giving semantics to a system by first defining a 'single-stepper' and defining then a function that runs the single-stepper a given number of steps. See, for example, [6].

[1] Note that Clark's completion axiom [8] expresses the negative part of an inductive definition in some cases, but not in general. Indeed the axioms
$(\textit{even}\ 0)$
$\forall x((\textit{even}\ x) \Rightarrow (\textit{even}\ (S\ (S\ x))))$
$\forall x((\textit{even}\ x) \Rightarrow ((x = 0) \vee \exists y((\textit{even}\ y) \wedge (x = (S\ (S\ y))))))$ (completion axiom)
characterize a unique set.

But let us define the empty set E as the smallest set such that if x is in E then x is in E. In this case, the axioms
$\forall x((E\ x) \Rightarrow (E\ x))$
$\forall x((E\ x) \Rightarrow \exists y((E\ y) \wedge (x = y)))$ (completion axiom)
do not characterize a unique set.

6 λ-calculus with Nameless Dummies

In the presentation of the calculus above, we have ignored variable renaming problems. These problems cannot be ignored in a formal specification and implementation. We have therefore used the notion of terms with nameless dummies introduces by de Bruijn [dB72]. In these terms, variables have no names and each occurrence of a variable is represented by a positive integer: the relative depth of its binder.

For instance the term $[T : Prop][f : (T \to T) \to T](f\ [y : T](f\ [x : T]y))$ is expressed by $[Prop][(1 \to 2) \to 2](1\ [2](2\ [3]2))$, so is the term $[U : Prop][g : (U \to U) \to U](g\ [a : U](g\ [b : U]a))$.

The definitions above are transformed into:

Definition: Term

The set of term is the smallest set such that:

- $Prop$ is a term,
- $Kind$ is a term,
- if n is a positive integer then n is a term,
- if t and t' are terms then $(t\ t')$ is a term,
- if t and t' are terms then $[t]t'$ is a term,
- if t and t' are terms then $(t)t'$ is a term.

Definition: Context

A *context* Γ is a list of which the elements are either terms (variable declarations) or pairs of terms (constant declarations), i.e., the set of contexts is the smallest set such that:

- $[\,]$ is a context,
- if Γ is a context and T a term then $\Gamma[T]$ is a context,
- if Γ is a context and t and T are terms then $\Gamma[t : T]$ is a context.

Definition: Substitution (and lift by one while you are at it)

- $Prop[n \leftarrow u] = Prop$,
- $Kind[n \leftarrow u] = Kind$,
- $n[n \leftarrow u] = \uparrow_1^{n-1} u$,
- $p[n \leftarrow u] = p$ (if $p < n$),
- $p[n \leftarrow u] = p - 1$ (if $n < p$),
- $(t\ t')[n \leftarrow u] = (t[n \leftarrow u]\ t'[n \leftarrow u])$,

- $[t]t'[n \leftarrow u] = [t[n \leftarrow u]]t'[n+1 \leftarrow u]$,
- $(t)t'[n \leftarrow u] = (t[n \leftarrow u])t'[n+1 \leftarrow u]$,

where the lifting function $\uparrow^k_n$ is defined by:

- $\uparrow^k_n Prop = Prop$,
- $\uparrow^k_n Kind = Kind$,
- $\uparrow^k_n p = p$ (if $p < n$),
- $\uparrow^k_n p = p + k$ (if $p \geq n$),
- $\uparrow^k_n (t\ t') = (\uparrow^k_n t\ \ \uparrow^k_n t')$,
- $\uparrow^k_n [t]t' = [\uparrow^k_n t] \uparrow^k_{n+1} t'$,
- $\uparrow^k_n (t)t' = (\uparrow^k_n t) \uparrow^k_{n+1} t'$.

Note that introducing de Bruijn indices makes the definition of substitution less intuitive since we need to use the lifting operator $\uparrow^k_n$. We have here an example in which the specification is as tricky as a program, and we could fail to express it correctly. An interesting problem is to find a specification of the substitution function with de Bruijn indices as intuitive as the one with explicit names.

Definition: $\beta\delta$-reduction

- $([T]t\ u)\ \rhd_\Gamma t[1 \leftarrow u]$
- if $t : T$ is the n^{th} element of Γ then $n \rhd_\Gamma \uparrow^n_1 t$,
- $t\ \rhd_\Gamma t$,
- if $t\ \rhd_\Gamma u$ and $u\ \rhd_\Gamma v$ then $t\ \rhd_\Gamma v$,
- if $t\ \rhd_\Gamma u$ and $t'\ \rhd_\Gamma u'$ then $(t\ t')\ \rhd_\Gamma (u\ u')$,
- if $t\ \rhd_\Gamma u$ and $t'\ \rhd_\Gamma u'$ then $[t]t'\ \rhd_\Gamma [u]u'$,
- if $t\ \rhd_\Gamma u$ and $t'\ \rhd_\Gamma u'$ then $(t)t'\ \rhd_\Gamma (u)u'$.

Definition: $\beta\delta$-convertibility

- if $t\ \rhd_\Gamma u$ then $t \equiv_\Gamma u$,
- if $t \equiv_\Gamma u$ then $u \equiv_\Gamma t$,
- if $t \equiv_\Gamma u$ and $u \equiv_\Gamma v$ then $t \equiv_\Gamma v$.

Definition: Typing rules

- [ ] is well-formed,

- if $\Gamma \vdash T : s$ and $s \in \{Prop, Kind\}$ then $\Gamma[T]$ is well-formed,
- if $\Gamma \vdash t : T$ then $\Gamma[t : T]$ is well-formed,
- if Γ is well-formed then $\Gamma \vdash Prop : Kind$,
- if Γ is well-formed and T is the n^{th} element of Γ then $\Gamma \vdash n :\uparrow_1^n T$,
- if Γ is well-formed and $t : T$ is the n^{th} element of Γ then $\Gamma \vdash n :\uparrow_1^n T$,
- if $\Gamma \vdash T : s$, $\Gamma[T] \vdash U : s'$, $s \in \{Prop, Kind\}$ and $s' \in \{Prop, Kind\}$ then $\Gamma \vdash (T)U : s'$,
- if $\Gamma \vdash T : s1$, $\Gamma[T] \vdash U : s2$, $\Gamma[T] \vdash t : U$, $s1 \in \{Prop, Kind\}$ and $s2 \in \{Prop, Kind\}$ then $\Gamma \vdash [T]t : (T)U$,
- if $\Gamma \vdash t : (T)U$ and $\Gamma \vdash u : T$ then $\Gamma \vdash (t\ u) : U[1 \leftarrow u]$,
- if $\Gamma \vdash t : T$, $\Gamma \vdash U : s$ and $T \equiv_\Gamma U$ and $s \in \{Prop, Kind\}$ then $\Gamma \vdash t : U$.

7 Formal Specification

We are now ready to give the (positive part of the) formal specification of the function *well-formed*. Here is an informal sketch of the mapping from the syntax of the Calculus of Constructions described above to the corresponding objects in the Nqthm logic.

- $Prop$ and $Kind$ are represented as the symbols `'prop` and `'kind`.
- The variable n (i.e., the positive integer n, a de Bruijn index) is represented as itself.
- $(t1\ t2)$ is represented as `(list 'apply t1 t2)`.
- $[t1]t2$ is represented as `(list 'lambda t1 t2)`.
- $(t1)t2$ is represented as `(list 'product t1 t2)`.
- A context is represented as a list. The empty context is represented as `nil`. Definitional elements of a context are represented with `(list 'constant ty te)` and other elements are represented with `(list 'variable ty)`.

We start with the lifting function $\uparrow_n^k$ and the substitution function.

```
(defn lift (n k c)
  (cond ((numberp c) (if (lessp c n) c (plus c k)))
        ((listp c)
         (list (car c)
               (lift n k (cadr c))
               (lift (if (equal (car c) 'apply) n (add1 n)) k (caddr c))))
        (t c)))

(defn subst (d n c)
  (cond ((numberp c) (cond ((equal c n) (lift 1 (sub1 n) d))
                           ((lessp c n) c)
```

```
                     (t (sub1 c))))
       ((listp c)
        (list (car c)
              (subst d n (cadr c))
              (subst d (if (equal (car c) 'apply) n (add1 n)) (caddr c))))
       (t c)))
```

We then axiomatize the (positive part of the) reduction and convertibility relations. We first declare the function `red` to be a function of three arguments. `(red env t1 t2)` expresses that `t1` reduces to `t2` in environment `env`.

```
(dcl red (env t1 t2))

(add-axiom red-beta (rewrite)
  (red env (list 'apply (list 'lambda u1 u2) u3) (subst u3 1 u2)))

(add-axiom red-delta (rewrite)
  (implies (equal (nth env n) (list 'constant ty te))
           (red env n (lift 1 n te))))

(add-axiom red-lambda (rewrite)
  (implies (and (red env t1 u1) (red (cons (list 'variable t1) env) t2 u2))
           (red env (list 'lambda t1 t2) (list 'lambda u1 u2))))

(add-axiom red-product (rewrite)
   (implies (and (red env t1 u1)
                 (red (cons (list 'variable t1) env) t2 u2))
            (red env (list 'product t1 t2) (list 'product u1 u2))))

(add-axiom red-apply (rewrite)
  (implies (and (red env t1 u1) (red env t2 u2))
           (red env (list 'apply t1 t2) (list 'apply u1 u2))))

(add-axiom red-refl (rewrite)
  (red env t1 t1))

(add-axiom red-trans (rewrite)
  (implies (and (red env t1 t2) (red env t2 t3)) (red env t1 t3)))
```

`(equiv en t1 t2)` expresses that `t1` is convertible to `t2` in environment `env`.

```
(dcl equiv (env t1 t2))

(add-axiom equiv-red (rewrite)
  (implies (red env t1 t2) (equiv env t1 t2)))

(add-axiom equiv-sym (rewrite)
  (implies (equiv env t1 t2) (equiv env t2 t1)))

(add-axiom equiv-trans (rewrite)
  (implies (and (equiv env t1 t2) (equiv env t2 t3))
           (equiv env t1 t3)))
```

At last we axiomatize the (positive part of the) typing predicate and the well-formedness predicate.

`(types env te ty)` expresses that `te` has type `ty` in environment `env`.

`(well-formed env)` expresses that the environment `env` is well-formed.

```
(dcl types (env term type))

(dcl well-formed (env))

(add-axiom empty (rewrite)
  (well-formed nil))

(add-axiom declaration (rewrite)
  (implies (and (well-formed env) (member s '(prop kind)) (types env ty s))
           (well-formed (cons (list 'variable ty) env))))

(add-axiom sort (rewrite)
  (implies (well-formed env) (types env 'prop 'kind)))

(add-axiom variable (rewrite)
  (implies (and (well-formed env) (equal (nth env n) (list 'variable ty)))
           (types env n (lift 1 n ty))))

(add-axiom product (rewrite)
  (implies (and (member s1 '(prop kind))
                (member s2 '(prop kind))
                (types env ty1 s1)
                (types (cons (list 'variable ty1) env) ty2 s2))
           (types env (list 'product ty1 ty2) s2)))

(add-axiom abstraction (rewrite)
  (implies (and (member s1 '(prop kind))
                (member s2 '(prop kind))
                (types env ty s1)
                (types (cons (list 'variable ty) env) ty2 s2)
                (types (cons (list 'variable ty) env) t2 ty2))
           (types env (list 'lambda ty t2) (list 'product ty ty2))))

(add-axiom application (rewrite)
  (implies (and (types env t1 (list 'product u1 u2)) (types env t2 u1))
           (types env (list 'apply t1 t2) (subst t2 1 u2))))

(add-axiom conversion (rewrite)
  (implies (and (types env te ty1) (types env ty2 s) (equiv env ty1 ty2))
           (types env te ty2)))

(add-axiom constdecl (rewrite)
  (implies (types env te ty)
           (well-formed (cons (list 'constant ty te) env))))

(add-axiom constant (rewrite)
```

```
  (implies (and (well-formed env) (equal (nth env n) (list 'constant ty te)))
           (types env n (lift 1 n ty))))
```

8 Program

We start the implementation with the normalization function.

```
(defn apply-list (x l)
  (if (nlistp l) x (apply-list (list 'apply x (car l)) (cdr l))))

(defn normal (flg env term stack clock)
  (cond
   ((equal flg 'list)
    (if (nlistp term)
        nil
      (cons (normal t env (car term) nil clock)
            (normal 'list env (cdr term) nil clock))))
   (t (cond
       ((zerop clock) (apply-list term stack))
       ((listp term)
        (let ((op (car term)) (c1 (cadr term)) (c2 (caddr term)))
          (cond
           ((equal op 'apply)
            (normal t env c1 (cons c2 stack) clock))
           ((equal op 'product)
            (apply-list
             (list op
                   (normal t env c1 nil clock)
                   (normal t (cons (list 'variable c1) env) c2 nil clock))
             stack))
           ((equal op 'lambda)
            (if (nlistp stack)
                (list op
                      (normal t env c1 nil clock)
                      (normal t (cons (list 'variable c1) env) c2 nil
                              clock))
              (normal t env (subst (car stack) 1 c2) (cdr stack)
                      (sub1 clock))))
           (t f))))
       ((numberp term)
        (let ((val (nth env term)))
          (if (and val (equal (car val) 'constant))
              (normal t env (lift 1 term (caddr val)) stack (sub1 clock))
            (apply-list term (normal 'list env stack nil (sub1 clock))))))
       ((equal term 'prop) (apply-list 'prop stack))
       ((equal term 'kind) (apply-list 'kind stack))
       (t f))))
  ((ord-lessp (cons (add1 clock) (count term)))))
```

Then we implement the function that computes a type of a term.

```
(defn check-term (env term clock)
  (cond
   ((numberp term)
    (let ((val (nth env term)))
      (if val (lift 1 term (cadr val)) f)))
   ((equal term 'prop) 'kind)
   ((or (nlistp term) (not (equal nil (cdddr term)))) f)
   (t (let ((op (car term)) (c1 (cadr term)) (c2 (caddr term)))
        (cond
         ((equal op 'apply)
          (let ((typ1 (check-term env c1 clock))
                (typ2 (check-term env c2 clock)))
            (if (and typ1 typ2)
                (let ((ntyp1 (normal t env typ1 nil clock))
                      (ntyp2 (normal t env typ2 nil clock)))
                  (if (and (equal 'product (car ntyp1))
                           (equal ntyp2 (cadr ntyp1)))
                      (subst c2 1 (caddr ntyp1))
                    f))
              f)))
         ((equal op 'lambda)
          (let ((typ1 (check-term env c1 clock))
                (typ2 (check-term (cons (list 'variable c1) env) c2 clock)))
            (if (and (member typ1 '(prop kind))
                     typ2
                     (not (equal typ2 'kind)))
                (list 'product c1 typ2)
              f)))
         ((equal op 'product)
          (let ((typ1 (check-term env c1 clock))
                (typ2 (check-term (cons (list 'variable c1) env) c2 clock)))
            (if (and (member typ1 '(prop kind)) (member typ2 '(prop kind)))
                typ2
              f)))
         (t f))))))
```

At last, here is the function that checks that a context is well-formed.

```
(defn check-item (env item clock)
  (let ((nature (car item)))
    (if (equal nature 'constant)
        (let ((typ (cadr item)) (val (caddr item)))
          (let ((ty2 (check-term env val clock)))
            (if (and (equal nil (cdddr item))
                     (check-term env typ clock)
                     ty2
                     (equal (normal t env ty2 nil clock)
                            (normal t env typ nil clock)))
                (cons item env)
              f)))
      (if (equal nature 'variable)
          (let ((typ (cadr item)))
```

```
            (if (and (equal nil (cddr item))
                     (member (check-term env typ clock) '(prop kind)))
                (cons item env)
              f))
        f))))

(defn check (env clock)
(if (nlistp env)
    (equal 'nil env)
    (and (check (cdr env) clock) (check-item (cdr env) (car env) clock))))
```

This implementation is very influenced by [Hue89].

9 Unproved Lemmas

We failed to prove three lemmas that are needed to prove soundness and that require the negative part of the inductive definitions.

```
(add-axiom subject-reduction (rewrite)
  (implies (and (types env t1 ty) (red env t1 t2))
           (types env t2 ty)))

(add-axiom type-type (rewrite)
  (implies (types env te ty)
           (or (equal ty 'kind) (types env ty 'prop) (types env ty 'kind))))

(add-axiom red-kind (rewrite)
  (equal (red env 'kind te) (equal te 'kind)))
```

10 Soundness

Given the foregoing axioms, definitions, and unproved lemmas, we can now check with Nqthm our soundness result:

```
(implies (check l clock) (well-formed l))
```

Events suitable for driving Nqthm to check this result are given in [4].

11 An Example

Using this system, we have checked a proof of Tarski's fixed point theorem. The object was obtained by using the system COQ. Nqthm can prove (in fact simply by running code for `check`) that `check` returns non-`F` on this formal proof object, given a clock argument of 7, and thus by the theorem above, it follows that that the proof object is well-formed.

12 Proving The Soundness of a Enhanced Conversion Function

When we want to apply a function f of type $A \to B$ to a value of type A' we need to check that the terms A and A' are convertible. In the program above we normalize the terms A and A' and we check that their normal forms are equal. When $A = F(a)$ and $A' = F(a')$ where F is a constant declared in the context $F := t$ and a and a' are convertible terms, we normalize the terms $(t\ a)$ and $(t\ a')$ and check that the normal forms are equal. In [Hue89] Huet has proposed a more evolved method, in which when we get the same constant as head-symbol of the terms A and A' we first try to check that the arguments of this function are pairwise convertible and only when this test fails expand the constants. Keeping the specification the same, we have proved the soundness of another program using this method.

Conclusions and Speculations

Thus far in our investigation, we have formally specified and implemented in the Nqthm logic a proof-checker for the Calculus of Constructions and we have proved the soundness of the implementation, assuming three unproved lemmas.

Several problems remain unsolved.

- Our specification of substitution is not very intuitive, since we define it through the lifting operator $\uparrow_n^k$.
- We need to find a way to express the negative part of the inductive definitions to be able to complete the proofs of the three admitted lemmas and to prove the completeness of our implementation.
- We need to understand how to define the normalization function without a 'clock' argument, for which we need to be able to express a function that terminates only when its arguments belong to some class (here, well-typed terms) and prove that in our definition this function is only used with such arguments. Because the class in question is characterized by the function we are trying to define, this problem is nontrivial for Nqthm.
- We need to understand how to strengthen Nqthm in order to be able to prove the termination of this unclocked normalization function, something non-trivial since, by Gödel's second incompleteness theorem, it can be proved neither in the Calculus of Constructions nor in weaker systems, such as the recursive fragment of the Nqthm logic. For example, given a concrete representation for a suitably large ordinal and given a primitive recursive 'less-than' relation on this representation, we could easily 'wire' the Nqthm system to permit induction up to that ordinal. However, we do not currently know of such an ordinal and representation.
- As an alternative to attempting to formalize an unclocked-version of the normalization function, we might instead reformulate the *well-formed* predicate to take both an environment and a finite sequence of reduction operations to perform. That is, we could retreat from defining a 'decision procedure' for well-formedness to being satisfied with merely defining a proof-checker for well-formedness.

- As another alternative, we could define `check` via `EVAL$`, an interpreter for the partial recursive functions.

References

[1] P. Aczel, An introduction to Inductive Definitions, *Handbook of Mathematical Logic*, J. Barwise (Ed.), North-Holland, 1977, pp. 739-782.

[2] Th. Altenkirch, A Formalization of the strong normalization proof for system F in LEGO, *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science 664, 1993, pp. 13-28.

[3] S. Berardi, Girard Normalization Proof in LEGO, Informal proceedings of the first workshop on logical frameworks, Antibes, 1990, pp. 65-75.

[4] R. Boyer, G. Dowek, Towards Checking Proof-Checkers, Manuscript, 1992.

[5] R. Boyer, J. Moore, A Computational Logic Handbook, Academic Press, 1988.

[6] R. Boyer, Y. Yu, Automated Correctness Proofs of Machine Code Programs for a Commercial Microprocessor, *Automated Deduction – CADE-11, Lecture Notes in Computer Science 607*, D. Kapur (Ed.), Springer-Verlag, 1992, pp. 416-430.

[7] N.G. de Bruijn, Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem, *Indagationes Mathematicae*, 34, 5, 1972, pp. 381-392.

[8] K. L. Clark, Negation as Failure, *Logic and Data Bases*, H. Gallaire and J. Minker (Eds.), Plenum Press, New York, 1978, pp. 293-322.

[9] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université Paris VII, 1985.

[10] Th. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[11] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, Ch. Paulin-Mohring, B. Werner, The Coq Proof Assistant User's Guide, V5.8, INRIA-Rocquencourt, ENS-Lyon, 1993.

[12] G. Huet, The Constructive Engine, *A Perspective in Theoretical Computer Science*, Commemorative Volume for Gift Siromoney, R. Narasimhan (Ed.), World Scientific Publishing, 1989.

[13] J. McKinna, R. Pollack, Pure Type Systems Formalized, *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science 664, 1993, pp. 289-305.

Infinite Objects in Type Theory

Thierry Coquand
Chalmers University

Work in progress, Comments wellcome

Abstract

We show that infinite objects can be constructively understood without the consideration of partial elements, or greatest fixed-points, through the explicit consideration of proof objects. We present then a proof system based on these explanations. According to this analysis, the proof expressions should have the same structure as the program expressions of a pure functional lazy language: variable, constructor, application, abstraction, case expression, and local let expression.

Introduction

The usual explanation of infinite objects relies on the use of greatest fixed-points of monotone operators, whose existence is justified by the impredicative proof of Tarski fixed point theorem. The proof theory of such infinite objects, based on the so called co-induction principle, originally due to David Park [18] and explained with this name for instance in the paper [15], reflects this explanation. Constructively, to rely on such impredicative methods is somewhat unsatisfactory and this paper is a tentative for a more direct understanding of infinite objects. Interestingly, the explicit consideration of proof objects plays an essential rôle here and this approach suggests an alternative reasoning system. In particular, the notion of constructors, or introduction rules, keeps the fundamental importance it has for proof system about well-founded objects [13], while it appears as a derived notion in proof systems based on co-induction (where this notion is secondary to the notion of destructors, or elimination rules). As a consequence, the strong normalisation property does not hold any more, but it is still the case that any closed term reduces to a canonical form.

Briefly, we can describe our approach as follows. A co-inductive predicate, relation, … is defined by its introduction rules. Following the proofs as programs principle, we represent them as constructors of a functional language with dependent types and each proof is now represented as a functional expression. Like in a programming language, we can define a function by recursion, which corresponds to a proof where the result we want to prove is used recursively. This cannot be considered to be a valid proof in general, and has to satisfy some conditions in order to be correct. We describe a simple syntactical check that ensures this correctness, which we believe leads to a natural style of proofs about infinite (or lazy) objects.

Since one important application we have in mind is the mechanisation of reasoning about programs and processes, we analyse in our formalism some concrete examples from the litterature [19, 15].

One possible way of reading this paper is to read the proof principle described in the subsection 1.2.4, and then to look at the examples in the second section. The first section contains motivations for and a purely inductive justification of this proof principle.

Besides to illustrate further the increasingly recognized importance of infinite proofs for programming language semantics, we hope to show also that the addition of infinite objects is an interesting extension of Type Theory. In particular, we can now represent a notion of processes in Type Theory.

1 General presentation

1.1 Type Theory of Well-Founded Objects

We recall briefly some basic notions of type theory of well-founded objects, that will be important for the extension to infinite objects. The books [4, 13] contain more detailed explanations, and the reference [3] describes the addition of case expressions and pattern-matching.

1.1.1 Semantics

A(n inductive) **set** A is defined by its **constructors**. A closed term of type A can be thought of as a well-founded tree, built out of constructors. We identify sets and **propositions**. The constructors can be interpreted as **introduction rules**, and a closed proof of the proposition A is a well-founded proof tree built out of introduction rules.

It is clear [13, 7], that, besides terms purely built out of constructors, one needs also to consider **noncanonical** expressions. The addition of such expressions is done in such a way however that any closed term of a closed set can be reduced to a **canonical form**, i.e. a term of the form $c(a_1, \ldots, a_n)$ where c is a constructor[1]. We can then associate in a natural way to any term a tree built out of constructors, and we require this tree to be well-founded. This tree is called the **computation tree** of a term. A **component** of a closed term is a (closed) term that appears in its computation tree. This defines an order relation of closed terms, called the **component ordering**.

What is essential is the fact that the component ordering is well-founded.

These notions can be traced back to Brouwer's idea of the "fully analysed" form of a proof [7].

1.1.2 Noncanonical Constants

We now give a general way of adding new noncanonical constant, which preserves this association of a well-founded proof tree to any closed object. This new constant f is first given a type $(x_1 : A_1, \ldots, x_n : A_n)A$, and we write its definition $f(x_1, \ldots, x_n) = e$, where e is an expression built on previously defined constants and case expressions. The definition may be recursive, but, using the semantics of a term as a well-founded tree, we can ensure that the recursive calls are well-founded and justify in such a way this recursivity. We notice, as in [6], that there is a simple syntactical check that ensures this: there exists a lexicographic ordering of the arguments of f, such that all recursive calls are well-founded for the lexicographic extension of the component ordering.

[1] Our notations will follow [4].

1.1.3 Examples

The set $\mathbf{N}$ of integers is defined by its constructors $\mathbf{0} : \mathbf{N}$ and $\mathbf{s} : (\mathbf{N})\mathbf{N}$. A closed element of type $\mathbf{N}$ is thus a finite object of the form $\mathbf{s}^k(\mathbf{0})$.

Let us consider a type $\mathbf{P}$ with constructors $\mathbf{out} : (\mathbf{N})(\mathbf{P})\mathbf{P}$, $\mathbf{in} : ((\mathbf{N})\mathbf{P})\mathbf{P}$ and $\mathbf{nil} : \mathbf{P}$. A closed element $p : \mathbf{P}$ has to be thought of as a well-founded tree built out of the constructors $\mathbf{out}$, $\mathbf{in}$ and $\mathbf{nil}$. For instance, if $u(n) = \mathbf{out}(n, \mathbf{nil})$, the term $\mathbf{in}(u)$ has for components, besides itself, all the instances $\mathbf{out}(\mathbf{s}^k(\mathbf{0}), \mathbf{nil})$, $\mathbf{s}^k(\mathbf{0})$ and $\mathbf{nil}$.

The requirement that we should be able to think of all closed elements as a tree, with a definite branching (that may be infinite), imposes strong restriction on the type of the constructors. Thus, we cannot have a set X with a constructor of type $((X)X)X$ or of type $(((X)N)N)X$. However, a condition of strict positivity [8] on the type of the constructors is enough to ensure that we can think of elements as trees built out of constructors.

The Ackerman function $A : (\mathbf{N})(\mathbf{N})\mathbf{N}$ defined by the equation

$$A(\mathbf{0}, n) = \mathbf{s}(n),\ A(\mathbf{s}(m), \mathbf{0}) = A(m, \mathbf{s}(\mathbf{0})),\ A(\mathbf{s}(m), \mathbf{s}(n)) = A(m, A(\mathbf{s}(m), n)),$$

follows the schema of definition, since the recursive calls are always smaller for the lexicographic ordering.

The equations above are a sugared form of the following definition

```
A(x,y) = case x of

                0     => s(y)

              | s(x') => (case y of

                                0     => A(x',s(0))

                              | s(y') => A(x',A(x,y'))

                          end)

            end
```

1.2 Infinite Objects

1.2.1 Analogy between proofs and processes

It is tempting to think of an object of type $\mathbf{P}$ as a process p which has three possible behaviours: it can either emit an integer and becomes p_1, when it is of the form $\mathbf{out}(n, p_1)$, or express that it needs an integer as input, if it is of the form $\mathbf{in}(f)$, or show that it is inert, if it is of the form $\mathbf{nil}$. In this reading, the computation tree of an element is the "behaviour tree" [16] of the process associated to it.

With this reading, the restriction to well-founded objects seems too strong. For the type $\mathbf{P}$ as defined above, this will mean that we consider only processes that eventually become inert.

This forbids for instance a process $p = \mathbf{in}([n]\mathbf{out}(\mathbf{s}(n), p))$ that interactively asks for an integer and outputs its successor.

It is thus quite natural to consider also **lazy** sets that are, like inductive sets, specified by their constructors, but whose elements can be thought of as arbitrary, not necessarilly well-founded, trees built out of constructors. We still require of noncanonical elements that they reduce eventually to constructor forms.

As we have seen, the consideration of such objects is common in the analysis of processes [16]. The consideration of not necessarilly well-founded objects arouse also in proof theory, for the study of proofs in ω-logic [11].

The process $p = \mathbf{in}([n]\mathbf{out}(\mathbf{s}(n), p))$ recursively defined is an element of the lazy set $\mathbf{P}$. It makes also sense of considering the lazy set Ω, which has only one constructor $s : (\Omega)\Omega$. The well-founded version of this type is empty, but the lazy set Ω contains the recursively defined element $\omega = s(\omega)$. An object is called **productive** if we can associate a computation tree to it, without requiring this computation tree to be well-founded. If x is a (productive) object of type Ω, it should reduce to an element $x = s(x_1)$ (because s is the only constructor of the set Ω), and similarly x_1 should reduce to an element $x_1 = s(x_2)$, and so on.

Though this notion of productivity seems clear, at least in the case of finitely branching trees, the main problem will be to give a finitary precise definition of productivity. We will give this definition after reviewing some attempts in adding infinite objects to type theory. Though simple, it is quite surprising that this definition achieves this goal without infinitary considerations based on greatest fixed-points or infinite ordinals[2].

1.2.2 Problem with the addition of infinite objects

Some problems in adding infinite objects in Type Theory are analysed in the reference [14]. One basic problem can be expressed as follows: how to add infinite objects without also adding partial objects, that is objects that do not reduce to a canonical form? For instance, it is not correct to define a function $f : (\Omega)\Omega$ by the equation $f(s(x)) = f(x)$, because $f(\omega)$ does not reduce then to canonical form. In contrast, the definition $f(s(x)) = s(f(x))$ should be clearly allowed, because the element $f(x)$ is then productive if $x : \Omega$ is productive.

Is their a simple syntactical criteria that ensures the preversation of productivity, which is not too restrictive?

In our analysis, a definition of the primitive recursive form $f(s(x)) = g(x, f(x))$ cannot be justified in general. Indeed, the justification of such a definition relies ultimately on the fact that we consider only well-founded objects [13]. In [14], a different view is followed, based on an unexpected analogy between the addition of infinite objects in type theory and non-standard extensions in non-standard analysis. This explanation rejects circular definitions such as $\omega = s(\omega)$, but allow non well-founded definitions such as

$$\omega_0 = s(\omega_1),\ \omega_1 = s(\omega_2), \ldots$$

In the next paragraph, we will suggest a proof principle, which can also be seen as a way of defining functions over not necessarilly well-founded objects, that relies directly on the semantics of an object as a not necessarilly well-founded tree built out of constructors.

[2] This definition can be extracted from the paper [11], where the notion of "convergence" corresponds to our notion of productivity.

1.2.3 A key example

At this point, the basic difficulty is to find a way of defining functions that ensures that any instances of such functions on productive elements are productive. For this, we need a precise notion of productivity.

In order to find this definition, let us analyse a key example. We consider the function $F : (\mathbf{P})\mathbf{P}$ defined by the equations

$$F(\mathbf{nil}) = \mathbf{nil}, \quad F(\mathbf{in}(u)) = \mathbf{in}([n]F(u(n))), \quad F(\mathbf{out}(n,p)) = \mathbf{out}(n, F(p)).$$

It should be clear intuitively that $F(p)$ is productive if p is productive. How can we be convinced of this fact in a clear and rigourous way? One answer may be a definition of productivity as a greatest fixed-point. While this answer is formally satisfactory, it can be argued that its impredicative use of Tarski's fixed point theorem is not a satisfactory finitary explanation of infinite objects.

It can be noticed however that it is directly clear that $F(p)$ reduces to a canonical form if p is productive. Furthermore, we can see that all components of $F(p)$ are then of the form $F(q)$, for some productive $q : \mathbf{P}$, or n for some $n : \mathbf{N}$. This suggests the following definition.

Definition: An element is **productive** iff all its components have a canonical form.

It is then clear that $F(p)$ is productive if p is productive. This will be directly generalized in the next section.

1.2.4 Proof principle for infinite objects

Let f be a constant of type $(x_1 : A_1, \ldots, x_n : A_n)A$ where A is a set, we define simulatenously when f is **guarded in** an expression e and when f is **authorized** in e (this second notion is only an auxiliary notion, and guarded implies authorized). This is by case on e :

- if f does not occur in e, then f is guarded and authorized in e,
- if e is of the form $c(u_1, \ldots, u_n)$ where c is a constructor, and f is authorized in all u_i, then f is guarded (and authorized) in e,
- if e is of the form $[x]u$, and f is guarded (resp. authorized) in u, then f is guarded (resp. authorized) in e,
- if e is a case expression $case(v, p_1 \to e_1, \ldots, p_n \to e_n)$, then f is guarded (resp. authorized) in e iff f does not occur in v and f is guarded (resp. authorized) in $e_1, \ldots, e_n$,
- if e is of the form $f(u_1, \ldots, u_n)$ and f does not occur in $u_1, \ldots, u_n$, then, f is authorized in $f(u_1, \ldots, u_n)$.

The importance of this notion comes from the following result, where we assume that all closed expressions that do not contain f are productive.

Lemma: if f is defined by $f(x_1, \ldots, x_n) = e$ and f is guarded in e, then all the components of e either does not contain f, or are of the form $f(u_1, \ldots, u_p)$ where f does not occur in $u_1, \ldots, u_n$.

Theorem: If $f : (x_1 : A_1, \ldots, x_n : A_n)A$ has a recursive definition $f(x_1, \ldots, x_n) = e$ where all recursive occurences of f in e are guarded by constructors, then $f(p_1, \ldots, p_n)$ is productive whenever $p_1, \ldots, p_n$ are.

Proof: We establish first that any closed instance $f(p_1, \ldots, p_n)$ reduces to a canonical form. By the previous lemma, it follows that any closed instance of f is productive[3]. **Q.E.D.**

This theorem can be read as a proof principle. In order to establish that a proposition ϕ follows from other propositions $\phi_1, \ldots, \phi_q$, it is enough to build a proof term e for it, using not only natural deduction and case analysis, but also using the proposition we want to prove recursively, provided such a recursive call is guarded by introduction rules. We hope to show by the examples given below that this reasoning principle is quite flexible and intuitive in practice.

1.2.5 Some remarks on this proof principle

First, it has to be noticed that this criteria cannot accept nested occurences of the function, contrary to the well-founded cases. Thus, we cannot define a function $f : (\Omega)\Omega$ by the equation

$$f(s(s(x))) = s(f(f(x))),$$

since the nested occurence of f in the right handside is not guarded. Indeed, in this case, it can be checked that $f(\omega)$ is not productive: it has for component $f(f(\omega))$ which does not reduce to canonical form.

Another remark is that we can combine this test with the previous test on well-founded recursive calls, if some arguments are ranging over well-founded types. This situation will occur in one example [15] analysed below, where an infinite proof is defined by well-founded recursion over an evaluation relation.

Finally, this guarded condition may seem too restrictive, especially in the definition of functions over infinite objects. Several programs on streams, even if they preserve productivity, do not obey in general this guarded condition [22]. But we think that the situation is similar to the one of well-founded objects, where the condition on structurally smaller recursive calls does not capture all usual definitions of programs defined over well-founded objects (though its scope is surprisingly large [3, 6]).

Though this does not seem to be the general case, some definitions that are not guarded can be turned easily in definitions that are guarded. For instance, if we consider the set of streams of integer $\mathbf{S}$ with only one constructor $\mathbf{cons} : (\mathbf{N})(\mathbf{S})\mathbf{S}$, we can define of the function $map : ((\mathbf{N})\mathbf{N})(\mathbf{S})\mathbf{S}$ by the guarded equation

$$map(f, \mathbf{cons}(x, l)) = \mathbf{cons}(f(x), map(f, l)),$$

and thus consider the equation

$$u = \mathbf{cons}(\mathbf{0}, map(\mathbf{s}, u)),$$

which should represent the stream $\mathbf{cons}(\mathbf{0}, \mathbf{cons}(\mathbf{s}(\mathbf{0}), \ldots))$. This definition is not allowed because it is not guarded. It can be replaced however by the definition of the function $v : (\mathbf{N})\mathbf{S}$

$$v(n) = \mathbf{cons}(n, v(\mathbf{s}(n))),$$

[3] The guarded condition is well-known for the recursive definition of processes [16]. The two important points here are first, its justification based on an inductive notion of productivity, and second, its use as a proof principle.

and the definition $u = v(\mathbf{0})$.

Furthermore, the first intended application is for reasoning about infinite objects, and not for programming on them. For this application, the guarded condition is enough to give a proof system at least as powerful as the one based on co-induction, and seems more flexible on the examples we have tried. It is actually by trying to understand intuitively what was going on in proofs by co-induction that the guarded condition came out as a proof principle.

To summarize, what is important about the guarded condition is that it can be ensured by a simple syntactical check, that it can be directly justified, and that it seems to provide a powerful enough proof principle for reasoning about infinite objects.

1.3 Reformulation with rule sets

In this section we express in an abstract way how one can understand inductively a greatest fixed-point. We follow the terminology of [1].

We start with a set U of atoms and a set Φ of **rules**, that are pairs (X, x) such that $X \subseteq U$ and $x \in U$. We write $\Phi : X \mapsto x$ to mean that $(X, x) \in \Phi$. An element $(X, x) \in \Phi$ is called a rule of **conclusion** x and of **premisses** X. There is a monotone operator ϕ associated to Φ, given by

$$\phi(Y) = \{x \in U \mid \Phi : X \mapsto x \text{ for } X \subseteq Y\}.$$

The **kernel** of ϕ is given by

$$K(\phi) = \bigcup\{X \mid X \subseteq \phi(X)\}.$$

This is the greatest fixed point of ϕ.

We now give a purely inductive description of $K(\phi)$ in the case where Φ is **deterministic**, i.e. when $\Phi : X_1 \mapsto x$ and $\Phi : X_2 \mapsto x$ entail $X_1 = X_2$.

First, we define $S_\Phi(x)$ as the set of $y \in U$ such that there exists $\Phi : X \mapsto x$ with $y \in X$. Let $z \in S^*_\Phi(x)$ mean that $z = x$ or inductively that $z \in S^*_\Phi(y)$ for some $y \in S_\Phi(x)$. An element of $S_\Phi(x)$ is called a **direct component** of x, and an element of $S^*_\Phi(x)$ a **component** of x. Let $C(\phi) \subseteq U$ be the set of $x \in U$ such that there exists a rule of conclusion x. This defines the set of **canonical** elements. The alternative description of $K(\phi)$ is

$$K'(\phi) = \{x \in U \mid S^*_\Phi(x) \subseteq C(\phi)\},$$

that is, $K'(\phi)$ is the set of elements whose components are all canonical.

Theorem: $K(\phi) = K'(\phi)$.

Proof: If $A \subseteq \phi(A)$ and $x \in A$, then we have $A \subseteq C(\phi)$ and $S^*_\Phi(x) \subseteq A$, using the fact that Φ is deterministic, and hence all the components of x are canonical. This shows the inclusion $K(\phi) \subseteq K'(\phi)$. Conversely, the inclusion $K'(\phi) \subseteq \phi(K'(\phi))$ holds in general, and hence $K'(\phi) \subseteq K(\phi)$, without any hypothesis on Φ. **Q.E.D**

2 Simple examples of proofs and programs

2.1 Divergence

We introduce the following set of expressions

$$\mathbf{0} : \mathbf{Exp}, \ \mathbf{s} : (\mathbf{Exp})\mathbf{Exp}, \ \omega : \mathbf{Exp},$$

and the following inductively defined relation

$$\mathbf{e}_1 : \mathbf{Eval}(\mathbf{0}, \mathbf{0}),\ \mathbf{e}_2 : (x : \mathbf{Exp})\mathbf{Eval}(\mathbf{s}(x), \mathbf{s}(x)),\ \mathbf{e}_3 : \mathbf{Eval}(\omega, \mathbf{s}(\omega))$$

and the following lazy predicate

$$\mathbf{i} : (x, y : \mathbf{Exp})(\mathbf{Eval}(x, \mathbf{s}(y)))(\mathbf{inf}(y))\mathbf{inf}(x).$$

The term

$$p_\infty : \mathbf{inf}(\omega)$$

is defined by the guarded equation

$$p_\infty = \mathbf{i}(\omega, \omega, \mathbf{e}_3, p_\infty),$$

and thus, it is a proof of $\mathbf{inf}(\omega)$.

Though this example is almost trivial, it illustrates the difference between the present proof system and proofs based on co-induction. A proof that ω is divergent using co-induction will consist in finding a predicate P, which holds for ω, such that $P(x)$ implies that there exists y such $\mathbf{Eval}(x, y)$ and $P(y)$. Thus, one has to find an "invariant" predicate. By contrast, the present approach does not involve the search of suitable predicates, but analyses the problem by looking at the introduction rule for the predicate **inf**. (We can see in this way that $\mathbf{inf}(\omega)$ has only one proof.)

2.2 Abstract divergence

In general, if we start with a set A with a binary relation R, one can describe inductively the subset of accessible elements

$$\mathbf{acc} : (x : A)((y : A)(R(x, y))\mathbf{Acc}(y))\mathbf{Acc}(x),$$

and the lazy subset of divergent elements

$$\mathbf{inf} : (x, y : A)(R(x, y))(\mathbf{Inf}(y))\mathbf{Inf}(x).$$

Classically, these subsets form a partition of A. In the present intuitionistic framework, one cannot expect in general to have a proof of $(x : A)[\mathbf{Acc}(x) + \mathbf{Inf}(x)]$.

In particular, we cannot derive in our system some results of [12], which establish the equivalence of two notions of divergence using the fact that an element either diverges or converges. It seems quite interesting to investigate this problem more in detail from an intuitionistic point of view (our guess is that this equivalence is not really used, and the non equivalence indicates only that the stronger notion of divergence is the correct intuitionistic notion).

It is however possible to show that these subsets are disjoint, by defining

$$\phi : (x : A)(\mathbf{Acc}(x))(\mathbf{Inf}(x))\mathbf{N}_0$$

with the following equation

$$\phi(x, \mathbf{acc}(x, f), \mathbf{inf}(x, y, q, r)) = \phi(y, f(y, q), r),$$

which is well-founded because the recursive call of ϕ is smaller on its second argument.

2.3 Representation of an unreliable medium

We want to build an element $m : \mathbf{P}$ that can be thought of as an unreliable medium: it asks first for an integer input, and either forgets it, or outputs it, and this recursively. For this, we introduce an infinite oracle set $\mathbf{C}$ with two constructors $0 : (\mathbf{C})\mathbf{C}$ and $1 : (\mathbf{C})\mathbf{C}$. An object of the set $\mathbf{C}$ can thus be thought of as an infinite stream of the form $0(1(0(0(\ldots))))$, and in this case, the computation tree of a term is similar to the binary development of a real number.

The following equations define a function $m : (\mathbf{C})\mathbf{P}$

$$m(0(\omega)) = \mathbf{in}([n]m(\omega)),\ m(1(\omega)) = \mathbf{in}([n]\mathbf{out}(n, m(\omega))),$$

since these equations satisfy the guarded condition.

What is important about this representation is that we will be able to define by a lazy predicate on $\mathbf{C}$ when an element of $\mathbf{C}$ contains infinitely many 1, and hence to specify when an unreliable medium is fair.

2.4 Definition of co-recursion

We now show on one example how to translate co-induction and co-recursion in our proof system. We suppose given a map $f : (X)[X + X]$ and we want to build from this a map $corec(f) : (X)\mathbf{C}$ satisfying the usual co-recursive equations [19]. For this, we define first $\phi : (X + X)\mathbf{C}$ by the guarded equations

$$\phi(\mathbf{inl}(x)) = 0(\phi(f(x)))\quad \phi(\mathbf{inr}(x)) = 1(\phi(f(x))).$$

One can then check that $corec(f)(x) = \phi(f(x))$ is such that $corec(f)(x) = 0(corec(f)(y))$ when $f(x)$ is of the form $\mathbf{inl}(y)$ and $corec(f)(x) = 1(corec(f)(y))$ when $f(x)$ is of the form $\mathbf{inr}(y)$. Hence, we have a representation of co-recursion over the lazy set $\mathbf{C}$.

This indicates how one can develop a realisability semantics of co-induction with streams (see [23]), in such a way that an element of a coinductive type is interpreted by a productive element.

2.5 Fairness

We introduce an inductively defined predicate $\mathbf{Event}_1$ on $\mathbf{C}$, such that $\mathbf{Event}_1(x, y)$ means that x is of the form $x = 0(0(\ldots 0(1(y))\ldots))$. We have two introduction rules

$$\mathbf{d}_1 : (x : \mathbf{C})\mathbf{Event}_1(1(x), x), \qquad \mathbf{e}_1 : (x, y : \mathbf{C})(\mathbf{Event}_1(x, y))\mathbf{Event}_1(0(x), y).$$

A proof of $\mathbf{Event}_1(x, y)$ has to be thought of as a finite term of the form

$$\mathbf{e}_1(x_1, y, \ldots, \mathbf{e}_1(x_{n-1}, y, \mathbf{d}_1(y))\ldots),$$

with $x = 0(x_1),\ x_1 = 0(x_2), \ldots, x_{n-1} = 1(y)$.

Using the inductively defined predicate $\mathbf{Event}_1$, we can now introduce the predicate $\mathbf{Inf}_1(x)$ which means that x contains infinitely many 1 in its development. It has only one introduction rule:

$$\mathbf{inf}_1 : (x, y : \mathbf{C})(\mathbf{Event}_1(x, y))(\mathbf{Inf}_1(y))\mathbf{Inf}_1(x),$$

and a proof of $\mathbf{Inf}_1(x_0)$ should be thought of as an infinite proof term of the form

$$\mathbf{inf}_1(x_0, x_1, p_1, \mathbf{inf}_1(x_1, x_2, p_2, \mathbf{inf}_1(\ldots)))$$

where p_n is a proof of $\mathbf{Event}_1(x_{n-1}, x_n)$. This corresponds closely to the intuition of what it means for such a stream to have infinitely many 1.

A fair unreliable medium will then be defined as a medium $m(\omega) : \mathbf{P}$, together with a proof of $\mathbf{Inf}_1(\omega)$.

2.6 Proof about the list of iterates

This example is taken from [19], and it is interesting to compare the proof given here to the co-inductive proof presented in this paper. We define first a lazy relation on the set of stream of integers with the only constructor

$$\mathbf{eq} : (n : \mathbf{N})(l_1, l_2 : \mathbf{S})(\mathbf{Eq}(l_1, l_2))\mathbf{Eq}(\mathbf{cons}(n, l_1), \mathbf{cons}(n, l_2)).$$

As a parenthesis, let us illustrate further our proof principle by showing that $\mathbf{Eq}$ is transitive. For this, we declare

$$trans : (l_1, l_2, l_3 : \mathbf{S})(\mathbf{Eq}(l_1, l_2))(\mathbf{Eq}(l_2, l_3))\mathbf{Eq}(l_1, l_3),$$

and define it by the guarded equation

$$\begin{aligned} &trans(\mathbf{cons}(n, l_1), \mathbf{cons}(n, l_2), \mathbf{cons}(n, l_3), \mathbf{eq}(n, l_1, l_2, p), \mathbf{eq}(n, l_2, l_3, q)) \\ &\qquad = \mathbf{eq}(n, l_1, l_3, trans(l_1, l_2, l_3, p, q)). \end{aligned}$$

We end this parenthesis, and present the problem: it is to show that, if we define $v : (\mathbf{N})\mathbf{S}$ by the guarded equation

$$v(n) = \mathbf{cons}(n, v(\mathbf{s}(n))),$$

and $map : ((\mathbf{N})\mathbf{N})(\mathbf{S})\mathbf{S}$ is defined by the guarded equation

$$map(f, \mathbf{cons}(n, l)) = \mathbf{cons}(f(n), map(f, l)),$$

then we have $\mathbf{Eq}(l_0, v(0))$, whenever $\mathbf{Eq}(l_0, \mathbf{cons}(0, map(\mathbf{s}, l_0)))$.

We introduce first the lazy relation $\mathbf{M}(l_1, l_2)$ on $\mathbf{S}$ with the only introduction rule

$$\mathbf{m} : (l_1, l_2 : \mathbf{S})(\mathbf{M}(l_1, l_2))(n : \mathbf{N})\mathbf{M}(\mathbf{cons}(n, l_1), \mathbf{cons}(\mathbf{s}(n), l_2)).$$

We can then define

$$g : (l : \mathbf{S})\mathbf{M}(l, map(\mathbf{s}, l)),$$

by the only guarded equation

$$g(\mathbf{cons}(x, l)) = \mathbf{m}(l, map(\mathbf{s}, l), x, g(l)).$$

Next we define a function

$$f : (n : \mathbf{N})(l, l' : \mathbf{S})(\mathbf{Eq}(l, \mathbf{cons}(n, l')))(\mathbf{M}(l, l'))\mathbf{Eq}(l, v(n))$$

by the guarded equation

$$f(n, \mathbf{cons}(n, l), \mathbf{cons}(\mathbf{s}(n), l'), \mathbf{eq}(n, l, l', p), \mathbf{m}(n, l, l', q)) = \mathbf{eq}(n, l, v(\mathbf{s}(n)), f(\mathbf{s}(n), l, l', p, q)).$$

This represents the following definition

```
f(n,l,l',p,q) =
 case p of

  eq(n1,l1,l1',p1) => case q of

                        m(n2,l2,l2',q1) =>
                              eq(n1,l1,v(s(n)),f(s(n),l1,l1',p1,q1))

                      end

 end
```

We have then

$$f(\mathbf{0}, l_0, map(\mathbf{s}, l_0), h, g(l_0)) : \mathbf{Eq}(l_0, v(0)) \quad [h : \mathbf{Eq}(l_0, \mathbf{cons}(0, map(\mathbf{s}, l_0)))].$$

2.7 Soundness of a type inference system

As a test example, we have represented the problem of soundness of a type inference system analysed in [15]. This corresponds to using the present version of type theory with possibly infinite objects instead of Peter Aczel non-well-founded set theory [2]. We suppose given a set of constants and we introduce

```
EXP : Set

const_exp : (CONST)EXP
lambda : (IDENT;EXP)EXP
app : (EXP;EXP)EXP
var : (IDENT)EXP
fix : (IDENT;IDENT;EXP)EXP

VAL : Set
ENV : Set

const_val : (CONST)VAL
clos : (IDENT;EXP;ENV)VAL
nil_env : ENV
cons_env : (IDENT;VAL;ENV)ENV
```

We can then define

```
loop : (x:IDENT;f:IDENT;exp:EXP;E:ENV)VAL
loop(x,f,exp,E) = clos(x,exp,cons_env(f,loop(x,f,exp,E),E))
```

which is guarded, and corresponds to the object cl_∞ described in [15].

We shall not describe the proof in detail, but only emphasize some points. The relation $v : \tau$ given by the rule (15) of the paper [15] is seen in our formalism as the introduction rule for a lazy relation between expressions and types. Thus, in the case of recursion, rule 6, page 217 (which is the only case where our proofs differ), we see the problem of proving $cl_\infty : \tau$ as the problem of building an infinite proof tree ending with $cl_\infty : \tau$. But this is direct, because $cl_\infty =< x, exp, E + \{f \mapsto cl_\infty\} >$ (we use the notation of [15]) and we can use recursively $cl_\infty : \tau$.

3 Mechanization

We now discuss the mechanization of the present system, that is the existence of an algorithm for type/proof-checking, and how to design an interactive proof search.

The main problem for type-checking is the type-checking of case expression, because this introduces an unification problem. Let us say that an expression is first-order iff it is built out of constructors and variables. If we impose that the conclusion of any introduction rule is first-order, then, we can use the first-order unification algorithm if we do only case analysis over variables whose types are first-order expressions. This restriction to first-order expression does not seem too strong in practice.

The checking of the well-foundedness condition, or the guarded condition, does not raise any problem. One alternative is that the user specifies himself a lexicographic ordering on the arguments, and then the system checks that all recursive calls that are not guarded are smaller for this ordering. The other alternative is that the system collects all recursive calls (all the non guarded recursive calls in the case of the definition of an infinite object), and tries to find by itself a suitable lexicographic ordering which ensures termination.

We believe that this system leads to an intuitive interactive proof system, well-suited for providing a mechanical help in the development of proofs in relational (or natural) semantics [15]. The user introduces new sets, predicates, relations defined by their introduction rule. We remark that, in practice and probably because it is clearer, in [12, 4], the lazy relations are not given by their elimination rules, but by their introduction rules.

When one wants to prove a result, or builds a noncanonical function, one first gives to it a name and a type. The use of case expression corresponds to the analysis of the hypotheses. By unification, this analysis generates subgoals that can be further analysed until we can write a solution. The possibility of declaring and proving local lemmas (that can be themselves recursively defined) corresponds to the addition of a local let construct.

Conclusion

Connections with the work of C. Raffalli [20], which presents ideas that seem similar in the logical system AF2 should be precised.

One main point of this paper, which goes back to the work of Lars Hallnäs [10], is that the infinitary notions that seem necessary in dealing with infinite objects, typically the use of greatest fixed-point or infinite ordinals, can be avoided altogether by explicit consideration of proof objects. Though it was not originally conceived with this remark in mind, the proof system we present can be seen as a further illustration of this basic idea[4].

LOOP DETECTION???

[4]One can see Martin-Löf's constructive explanation of the addition of non-standard elements through the explicit consideration of non-standard proof-objects [14] as yet another example of this idea.

Acknowledgement

The initial idea of this paper came through a discussion with Eike Ritter and Peter Dybjer about proof systems for infinite objects, and was precised by several discussions with Peter Dybjer, Andy Moran and Lars Hallnäs.

References

[1] P. Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, 739 - 782, (1977), Elsevier.

[2] P. Aczel. *Non-Well-Founded Set Theory* CSLI Lecture Notes, Vol. 14 (LSCI, Stanford, 1988)

[3] Th. Coquand. Pattern-Matching in Type Theory. Proceedings of the B.R.A. meeting on Proof and Types, (1992) Bastad.

[4] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretation. POPL'91, (1991).

[5] H. Curry and R. Feys. *Combinatory Logic, Vol. 1.* North-Holland Publishing Company

[6] O. Dahl. *Verifiable Programming.* Prentice-Hall, 1992.

[7] M. Dummett. *Elements of Intuitionism.* Oxford University Press, 1977.

[8] P. Dybjer. Inductive Families To appear in Formal Aspects of Computing (1993)

[9] J.Y. Girard. *Proof Theory and Logical Complexity.* Bibliopolis, 1988.

[10] L. Hallnäs. An Intensional Characterization of the Largest Bissimulation. Theoretical Computer Science 53 (1987), 335 - 343.

[11] L. Hallnäs. On the syntax of infinite objects: an extension of Martin-Löf's theory of expressions. LNCS 417, COLOG-88, P. Martin-Löf and G. Mints Eds., (1989), 94 - 103

[12] J. Hugues and A. Moran. A semantics for locally bottom-avoiding choice. Proceedings of the Glasogow Functional Programming Workshop'92, WICS (1992).

[13] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[14] P. Martin-Löf. Mathematics of Infinity. LNCS 417, COLOG-88, P. Martin-Löf and G. Mints Eds., (1989), 146 - 197

[15] R. Milner, M. Tofte. Co-induction in Relational Semantics Theoretical Computer Science 87 (1991), 209 - 220.

[16] R. Milner. *A Calculus of Communicating Systems* Report ECS-LFCS-86-7, Computer Science Department, University of Edingburg, 1986.

[17] B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[18] D. Park. Concurrency and automata on infinite sequences. in P. Deussen, editor, Proceedings of the 5th GI-conference on Theoretical Computer Science, LNCS 104, (1981), 167 - 183.

[19] L. Paulson. Co-induction and Co-recursion in Higher-order Logic. Draft (1993), University of Cambridge.

[20] C. Raffalli. Fixed points and type systems (Abstract) proceeding of the third B.R.A. meeting on Proofs and Types (1992), Bastad, 309.

[21] W. de Roever. On Backtracking and Greatest Fixpoints Formal Description of Programming Concepts, J. Neuhold (ed.), North-Holland, (1978), 621 - 639.

[22] B.A. Sijtsma. On the productivity of recursive list functions. ACM Transactions on Programming Language and Systems, Vol. 11, No 4 (1989), 633 - 649.

[23] M. Tatsuta. Realisability Interpretation of Coinductive Definitions and Program Synthesis with Streams. Proceedings of International Conference on Fifth Generation Computer Systems (1992) 666 - 673.

Intuitionistic Model Constructions and Normalization Proofs

Thierry Coquand and Peter Dybjer

Abstract

We investigate semantical normalization proofs for typed combinatory logic and weak λ-calculus. One builds a model and a function 'quote' which inverts the interpretation function. A normalization function is then obtained by composing quote with the interpretation function.

Our models are just like the intended model, except that the function space includes a syntactic component as well as a semantic one. We call this a 'glued' model because of its similarity with the glueing construction in category theory. Other basic type constructors are interpreted as in the intended model. In this way we can also treat inductively defined types such as natural numbers and Brouwer ordinals.

We also discuss how to formalize λ-terms, and show how one model construction can be used to yield normalization proofs for two different typed λ-calculi – one with explicit and one with implicit substitution.

The proofs are formalized using Martin-Löf's type theory as a meta language and mechanized using the ALF interactive proof checker. Since our meta language is intuitionistic, any normalization function is a normalization algorithm. Moreover, our algorithms can be seen as optimized versions of normalization proofs by the reducibility method, where the parts of the proof which play no role in returning a normal form are removed.

1 Introduction

There is a striking analogy between computing a program and assigning semantics to it. This analogy is for instance reflected in the similarity between the equations defining the denotional semantics of a language and the rules of evaluation in an environment machine [17] [1].

In this paper we use this analogy to give a semantical treatment of normalization for simply typed combinators and λ-calculus with weak reduction. The method consists of building a non-standard model, and a function ('quote') which maps a semantic object to a normal term representing it.

Our approach is strongly inspired by two early papers by Martin-Löf, where he emphasized the importance of intuitionistic abstractions on the meta level and the notion of definitional equality [19] and proved normalization for his type theory by using a model construction [20].

We pursue these ideas further. In particular we wish to emphasize the following aspects of our normalization proofs:

[1] The fundamental importance of this analogy was stressed by Per Martin-Löf in a recent talk about a substitution calculus.

- They are expressed as properties of normalization algorithms, rather than as the usual $\forall\exists$-propositions referring to binary reduction relations.

- The normalization algorithms are obtained by 'program extraction' from standard normalization proofs using the reducibility method. (Since any constructive proof is an algorithm it would be better to talk about optimization of a proof seen as a program.) The resulting models are simplifications of Martin-Löf's [19].

- The model constructions can be nicely expressed in the framework of initial algebra semantics (formalized in our constructive setting). We also discuss the role of definitional equality in this context [19].

- The models can be thought of as 'glueing syntax with semantics' a technique analogous to glueing (or sconing, or Freyd cover) in category theory [16, 22]. This technique has also been used by Lafont [15] for proving a coherence result for categorical combinators.

- Syntactic properties, such as Church-Rosser, may be replaced by semantic ones, such as the property that two terms are convertible iff their semantics are equal in the glued model.

- Martin-Löf's type theory is used as a formal meta language.

- The proofs are implemented on a machine using the interactive proof checker ALF.

We first develop a proof for pure typed combinatory logic (or positive implicational calculus). Then we extend it to full propositional logic, and note that all connectives except implication are interpreted as in the intended model. Finally, we show that the proof extends directly to inductively defined types, such as the natural numbers and Brouwer ordinals.

A very similar method also works for simply typed λ-calculus with weak reduction, where no reduction under λ is allowed. Here we focus on the representation problem for λ-terms. In particular we build a glueing model from normal λ-terms and meanings, and use it for normalization of two different versions of the λ-calculus. One of these has explicit substitutions and is similar to the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien, and Lévy [1]. The other is a nameless variant of the calculus used by Martin-Löf [20].

In an accompanying paper, a similar technique is used by Catarina Coquand [6] for normalization in simply typed λ-calculus with full reduction. In this case a Kripke model is used for the non-standard semantics.

2 Type theory and ALF

2.1 Martin-Löf's type theory

The formal meta language is Martin-Löf's type theory with inductive definitions and pattern matching. We use the intensional version of type theory formulated by Martin-Löf 1986, see Nordström, Petersson, and Smith [4].

The core of this language is the *theory of logical types* (*Martin-Löf's logical framework*). This is a dependently typed $\lambda\beta\eta$-calculus with a base type *Set* and a base type A for each object $A : Set$. We use the notation

$$(x_0 : \alpha_0; \ldots; x_n : \alpha_n)\alpha$$

for the type of n-ary functions[2];

$$[x_0, \ldots, x_n]a$$

for n-ary function abstraction; and

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

for n-ary function application.

We then use this framework for defining new sets and families of sets (*Martin-Löf's set theory*). These are defined by their constructors (or *introduction rules*). The rules follow Dybjer [11, 12], who gave natural deduction formulation of a class of admissible such *inductive definitions* in type theory. (See also Coquand and Paulin [8] for similar ideas in the context of the Calculus of Constructions.) This includes all standard set formers of Martin-Löf's type theory. Here we also introduce sets of (object language) types, families of sets of terms, families of conversion relations, etc. These are ordinary inductive definitions. In addition we need to define non-standard ordinals in the glued model by a generalized inductive definition.

To highlight the relationship between corresponding notions on the meta level and the object level we use certain notational conventions illustrated by the following table:

meta language	object language
$\rightarrow$	$\dot{\rightarrow}$
λ	$\dot{\lambda}$
app	$\mathtt{app}$

There are at least two notions of equality in type theory: *definitional equality*, written $a = b : A$, and *intensional equality*, written $I(A, a, b)$ (we often drop A in either case). Definitional equality is decidable and expressed by the equality *judgement*. Two terms are definitionally equal iff they are convertible iff they can be reduced to the same normal form by unfolding (possibly recursive) definitions. Intensional equality is expressed on the *propositional* level; it is a binary relation which is inductively defined by the reflexivity rule

$$ref : (A : Set; a : A)I(A, a, a)$$

If we have $a = b : A$, then $ref(a)$ is a proof of $I(A, a, b)$ by substitutivity of definitional equality. Conversely, if we have a closed proof of $I(A, a, b)$, where $a, b : A$ are closed terms, then $a = b : A$. This can be justified by appealing to the semantics of type theory: a closed proof of $I(A, a, b)$ should reduce to a closed canonical proof $ref(u)$ with $u = a = b : A$.

Furthermore, we introduce new functions by *pattern matching* following the ideas of Coquand [7]. This facility provides both a convenient notation and a useful generalization of the standard elimination rules or primitive recursive schemata [12] that can be derived from the introduction rules for a set.

Firstly, it allows the definition of functions by case analysis on several arguments simultaneously and uses a criterion that recursive calls must refer to *structurally smaller* arguments to ensure termination.

Secondly, unification is used to generate possible cases. This entails a strengthening of case analysis for inductively defined families. An example is that the proof of

$$peano4 : (I(0, 1))\emptyset$$

[2] An x_i which is not referred to by later types may be omitted.

now follows directly by pattern matching. The introduction rule for equality is reflexivity, and since this rule cannot be unified with $I(0,1)$ no cases are generated.

In our proofs we have used pattern matching in a non-essential way, and we shall indicate below how these uses can be reduced to standard primitive recursive schemata.

2.2 ALF - an interactive proof checker

We have implemented our proofs using the interactive proof checker ALF (Another Logical Framework) developed by Augustsson, Coquand, Magnusson, and Nordström. ALF is an implementation of Martin-Löf's logical framework. It is easy to introduce new constants and computation rules, for example, the formation and introduction rules for new sets and the typing and computation rules for functions defined by pattern matching. ALF implements the generation of cases described by Coquand [7], but the checks that recursive calls refer to structurally smaller arguments and that formation and introduction rules have the correct form have to be done manually.

The terms we present are edited versions of terms generated by the machine. The main difference is the use of implicit arguments (when they are clear from the context; this facility is not part of the ALF implementation yet) and overloading for making the notation more readable.

3 Typed combinatory logic

We first discuss pure typed combinatory logic, which by the Curry-Howard identification corresponds to a Hilbert-style axiomatization of positive implicational calculus. We give its syntax, its intended semantics, a normalization algorithm, and a correctness proof for this algorithm. We discuss algebraic and categorical aspects, and also how the algorithm can be extracted from a more standard proof using the reducibility method.

We then extend the approach to new type constructors which yield full intuitionistic propositional calculus. Finally, we consider inductively defined types, such as natural numbers and Brouwer ordinals.

3.1 Pure typed combinatory logic

3.1.1 Syntax

The identification of propositions and types strongly suggests to choose a formulation à la Church rather than à la Curry [3]. In the the former terms appear as proof objects of a Curry-Howard interpretation of the positive implicational calculus. The formulation à la Curry introduces redundant information.

We begin by defining the set of types inductively. The formation and introduction rules are

$$\mathtt{Type}^3 : Set$$

$$\dot{\rightarrow} \ : \ (\mathtt{Type}; \mathtt{Type})\mathtt{Type}$$

[3] We use the word 'type' and the formal name `Type` for 'type expression'. It would be less convenient but more consistent with the meta language to talk about 'set expression' and use the formal name `Set`.

We introduce more types below; at this point the reader may wish to add an uninterpreted base type $o : \mathtt{Type}$.

The terms form an inductively defined family $\mathtt{T}$ of sets indexed by types. The formation and introduction rules are

$$\mathtt{T} : (\mathtt{Type})Set$$

$$\begin{array}{rcl}
\mathtt{K} & : & (A, B : \mathtt{Type})\mathtt{T}(A \dot{\to} B \dot{\to} A) \\
\mathtt{S} & : & (A, B, C : \mathtt{Type})\mathtt{T}((A \dot{\to} B \dot{\to} C) \dot{\to} (A \dot{\to} B) \dot{\to} A \dot{\to} C) \\
\mathtt{app} & : & (A, B : \mathtt{Type}; \mathtt{T}(A \dot{\to} B); \mathtt{T}(A))\mathtt{T}(B)
\end{array}$$

3.1.2 Intended semantics

It is now possible to relate object language and meta language notions by giving the *intended* semantics. This is Tarski style semantics with intuitionistic notions on the meta level. What may be a little confusing at first here is that both meta level and object level are formalized: it is an interpreter (for terms of base type) written in ALF.

We define the interpretation of $\mathtt{Type}$ by recursion:

$$[\![\ ]\!] : (A : \mathtt{Type})Set$$

$$[\![A \dot{\to} B]\!] = [\![A]\!] \to [\![B]\!]$$

The interpretation of $\mathtt{T}$ is also by recursion:

$$[\![\ ]\!] : (A : \mathtt{Type}; a : \mathtt{T}(A))[\![A]\!]$$

$$\begin{array}{rcl}
[\![\mathtt{K}]\!] & = & \lambda([x]\lambda([y]x)) \\
[\![\mathtt{S}]\!] & = & \lambda([g]\lambda([f]\lambda([x]app(app(g, x), app(f, x))))) \\
[\![\mathtt{app}(f, a)]\!] & = & app([\![f]\!], [\![a]\!])
\end{array}$$

Note that this is structural recursion on an inductively defined family of sets.

Martin-Löf [19] discusses an intuitionistic notion of model and argues that equality (conversion) in the object language should be interpreted as definitional equality in the model. This requirement is satisfied for our (formalized) intended model in the following sense.

Firstly, we have the remarkable definitional equalities:

$$\begin{array}{rcl}
[\![\mathtt{app}(\mathtt{app}(\mathtt{K}, x), y)]\!] & = & [\![x]\!] \\
[\![\mathtt{app}(\mathtt{app}(\mathtt{app}(\mathtt{S}, x), y), z)]\!] & = & [\![\mathtt{app}(\mathtt{app}(x, z), \mathtt{app}(y, z))]\!]
\end{array}$$

The meta language expressions on both sides have the same normal form.

Moreover, we can introduce conversion as a family of inductively defined relations indexed by the types:

$$\mathtt{I}(,) : (A : \mathtt{Type}; a, a' : \mathtt{T}(A))Set$$

$$
\begin{array}{rcl}
\texttt{convK} & : & (A, B : \texttt{Type}; a : \texttt{T}(A); b : \texttt{T}(B))\texttt{I}(\texttt{app}(\texttt{app}(K, a), b), a) \\
\texttt{convS} & : & (A, B, C : \texttt{Type}; g : \texttt{T}(A \dot{\to} B \dot{\to} C); f : \texttt{T}(A \dot{\to} B); a : \texttt{T}(A)) \\
& & \texttt{I}(\texttt{app}(\texttt{app}(\texttt{app}(\texttt{S}, g), f), a), \texttt{app}(\texttt{app}(g, a), \texttt{app}(f, a))) \\
\texttt{convapp} & : & (A, B : \texttt{Type}; c, c' : \texttt{T}(A \dot{\to} B); a, a' : \texttt{T}(A); \texttt{I}(c, c'); \texttt{I}(a, a'))\texttt{I}(\texttt{app}(c, a), \texttt{app}(c', a')) \\
\texttt{convref} & : & (A : \texttt{Type}; a : \texttt{T}(A))\texttt{I}(a, a) \\
\texttt{convsym} & : & (A : \texttt{Type}; a, b : \texttt{T}(A); \texttt{I}(a, b))\texttt{I}(b, a) \\
\texttt{convtrans} & : & (A : \texttt{Type}; a, a', a'' : \texttt{T}(A); \texttt{I}(a, a'); \texttt{I}(a', a''))\texttt{I}(a, a'')
\end{array}
$$

We can prove by induction on $\texttt{I}(,)$ that

$$(A : \texttt{Type}; a, a' : \texttt{T}(A); \texttt{I}(a, a'))I([\![a]\!], [\![a']\!])$$

This proof is almost immediately mechanizable since most of the work is done by ALF's normalization. By the discussion of equality in section 2.1, we know that we can reexpress this proposition as the meta level statement that if $\texttt{I}(a, a')$ (under no assumptions), then the definitional equality $[\![a]\!] = [\![a']\!]$ follows.

3.1.3 Normalization algorithm

It is impossible to invert the interpretation function for the intended model, but if we enrich the interpretation of $\dot{\to}$ so that it has both a syntactic and a semantic component this becomes possible.

First we define a new interpretation function for types[4] [5]

$$[\![A \dot{\to} B]\!] \quad = \quad \texttt{T}(A \dot{\to} B) \times ([\![A]\!] \to [\![B]\!])$$

From this model we can retrieve normal forms:

$$\mathbf{q} : (A : \texttt{Type}; [\![A]\!])\texttt{T}(A)$$

$$\mathbf{q}_{A \dot{\to} B}(\langle c, f \rangle) \quad = \quad c$$

This is a first simple use of pattern matching with dependent types: we analyze both the first (implicit) and the second argument (which depends on the first one) simultaneously. Here it is straightforward to transform this definition into a standard primitive recursive schema (using higher types):

$$\mathbf{q}_{A \dot{\to} B}(p) \quad = \quad \mathit{fst}(p)$$

The interpretation of terms becomes:

$$[\![\ ]\!] : (A : \texttt{Type}; \texttt{T}(A))[\![A]\!]$$

[4] Alternatively, we could have introduced sets of normal terms $\texttt{Nt}(A)$ and used them as the syntactic component of the model instead. In that way we would formally ensure that the retrieved term is normal. This may be a technical advantage: for example one can use essentially the same set of normal terms for different kinds of λ-calculi, see section 4.2.1.

[5] Base types are interpreted syntactically: $[\![o]\!] = \texttt{T}(o)$

$$
\begin{aligned}
[\![\mathtt{K}]\!] &= \langle \mathtt{K}, \lambda([p]\langle \mathsf{app}(\mathtt{K}, \mathbf{q}(p)), \lambda([q]p)\rangle\rangle \\
[\![\mathtt{S}]\!] &= \langle \mathtt{S}, \lambda([p]\langle \mathsf{app}(\mathtt{S}, \mathbf{q}(p)), \lambda([q]\langle \mathsf{app}(\mathsf{app}(\mathtt{S}, \mathbf{q}(p)), \mathbf{q}(q)), \lambda([r]\, app_M\,(app_M\,(p,r), app_M\,(q,r))\rangle\rangle\rangle\rangle\rangle \\
[\![\mathsf{app}(f,a)]\!] &= app_M\,([\![f]\!], [\![a]\!])
\end{aligned}
$$

where we have used the following application operator in the model:

$$app_M : (A, B : \mathtt{Type}; [\![A \dot{\rightarrow} B]\!]; [\![A]\!])[\![B]\!]$$

$$app_M\,(\langle c, f\rangle, q) \;=\; app(f, q)$$

Finally, the normal form is extracted:

$$\mathbf{nf} : (A : \mathtt{Type}; \mathtt{T}(A))\mathtt{T}(A)$$

$$\mathbf{nf}(a) \;=\; \mathbf{q}([\![a]\!])$$

Also in this model we have the definitional equalities:

$$
\begin{aligned}
[\![\mathsf{app}(\mathsf{app}(\mathtt{K}, x), y)]\!] &= [\![x]\!] \\
[\![\mathsf{app}(\mathsf{app}(\mathsf{app}(\mathtt{S}, x), y), z)]\!] &= [\![\mathsf{app}(\mathsf{app}(x, z), \mathsf{app}(y, z))]\!]
\end{aligned}
$$

and

$$(A : \mathtt{Type}; a, a' : \mathtt{T}(A); \mathtt{I}(a, a'))I([\![a]\!], [\![a']\!])$$

and hence

$$(A : \mathtt{Type}; a, a' : \mathtt{T}(A); \mathtt{I}(a, a'))I(\mathbf{nf}(a), \mathbf{nf}(a'))$$

so that the rules of conversion are sound for the normal form semantics.

From this model we can also easily extract the model of normal terms.

As pointed out to us by Thorsten Altenkirch, this program can be translated into ML, and provides then a concise and elegant implementation of combinatory reduction.

```
datatype tm = s | k | ap of tm*tm;
datatype vl = v_arr of tm*(vl->vl);

fun term_part (v_arr (M,_)) = M;
fun val_ap (v_arr (_,f),x) = f x;

fun eval s =
  v_arr (s,fn x => v_arr (ap(s,term_part x),
                         fn y => v_arr (ap (ap (s,term_part x),term_part y),
                                        fn z => val_ap (val_ap (x,z),
                                                        val_ap (y,z)))))
  | eval k = v_arr (k,fn x => v_arr (ap(k,term_part x),
                                     fn y => x))
  | eval (ap(x,y)) = val_ap (eval x,eval y);

fun norm t = term_part (eval t);
```

3.1.4 Normalization proof

We shall prove that our normalization algorithm is correct in the sense that

$$(A : \texttt{Type}; a : \texttt{T}(A))\,\texttt{I}(a, \mathbf{nf}(a)).$$

Together with the fact that convertible terms have equal normal forms, which we proved above, and the fact that intensional equality is decidable, this yields a decision algorithm for convertibility.

According to this view correctness amounts to showing that two *syntactic* notions are equivalent: convertibility and equality of normal forms. But there is also a weaker but more fundamental form of *semantic* correctness: the normalization algorithm is correct since it preserves semantical equality. (This may be either intensional or extensional equality of elements in the intended model.)

Normalization yields an incomplete decision procedure for semantic equality. This is simply because weak conversion fails to distinguish between ξ and η-convertible terms and these conversions preserve the intended semantics. However it is complete for the glued semantics that we present below.

The correctness proof uses a construction closely related to glueing from categorical logic. Recall that function spaces in the model have a syntactic and a semantic component. We shall require that these components are coherent with each other: they are correctly 'glued' together in the sense that quotation commutes with application. This point will be clarified and expanded in the next section.

We introduce the property Gl to formalize this:

$$Gl : (A : \texttt{Type}; [\![A]\!])Set$$

$$Gl_{A \dot{\to} B}(\langle c, f\rangle) \quad = \quad \Pi([\![A]\!], [p](Gl_A(p) \to (Gl_B(app(f,p)) \times (\texttt{I}(\mathsf{app}(c, \mathbf{q}(p)), \mathbf{q}(app(f,p)))))))$$

We can now show that each term is interpreted as a glued value:

$$gl : (A : \texttt{Type}; a : \texttt{T}(A))\, Gl([\![a]\!])$$

$$\begin{aligned} gl(\texttt{K}) &= \lambda([p]\lambda([x]\langle\lambda([q]\lambda([y]\langle x, \texttt{convK}\rangle)), \texttt{convref}\rangle)) \\ gl(\texttt{S}) &= (\textit{long term}) \\ gl(\mathsf{app}(c,a)) &= app_{Gl}([\![c]\!], gl(c), [\![a]\!], gl(a)) \end{aligned}$$

where

$$app_{Gl} : (A, B : \texttt{Type}; q : [\![A \dot{\to} B]\!]; y : Gl(q); p : [\![A]\!]; x : Gl(p))\, Gl(app_M(q,p))$$

$$app_{Gl}(q, y, p, x) = \mathit{fst}(app(app(y,p), x))$$

From this it follows easily by T-recursion that

$$(A : \texttt{Type}; a : \texttt{T}(A))\,\texttt{I}(a, \mathbf{nf}(a))$$

This is an *external* proof of correctness in the sense that we first write the program **nf**, and then we make a logical comment on it. Note that in this way we do T-recursion (induction) *three* times: one for the program, one for glueing and one for convertibility. Alternatively, we could directly prove by *one* T-recursion that

$$(A : \texttt{Type}; a : \texttt{T}(A))\Sigma([\![A]\!], [p]\,Gl(p)\times(\texttt{I}(a, \mathbf{q}(p))))$$

This would be an *integrated* proof of correctness. If we only care about the normal form, however, it contains parts which are irrelevant. If the program is optimized by removing these parts we get **nf** back.

3.1.5 Algebraic view of the normalization proof

One can present the normalization proof algebraically by observing that

- the algebra of combinatory terms is initial among typed combinatory algebras;
- the model of glued values is another typed combinatory algebra;
- the interpretation function mapping each term to a glued element (that is a value in the model and a proof that it is glued) is the unique homomorphism from the initial algebra;
- the quote function (restricted to glued values) is a homomorphism back to the initial algebra (but it is not a morphism on the original algebra of combinatory terms).

Hence the normalization function is a homomorphism on the initial algebra. Hence it is (extensionally) equal to the identity. Since equality on terms is convertibility, this means that the normalization function preserves convertibility.

We shall now formalize these points in type theory using the fact that we can express model notions as *contexts* and instances of such notions as *explicit substitutions*.

Firstly, the *signature* Σ_{TCL} of typed combinatory algebras is represented by the following context:

$$\begin{array}{rcl} [M & : & (A:\texttt{Type})Set; \\ K_M & : & (A,B:\texttt{Type})M(A\dot{\to}B\dot{\to}A); \\ S_M & : & (A,B,C:\texttt{Type})M((A\dot{\to}B\dot{\to}C)\dot{\to}(A\dot{\to}B)\dot{\to}A\dot{\to}C); \\ app_M & : & (A,B:\texttt{Type}; M(A\dot{\to}B); M(A))M(B)] \end{array}$$

The intended model of these axioms is represented by the following substitution:

$$\begin{array}{rcl} [M & := & [\![\ ]\!]; \\ K_M & := & [A,B]\lambda([x]\lambda([y]x)); \\ S_M & := & [A,B,C]\lambda([g]\lambda([f]\lambda([a]\,app\,(app(g,a), app\,(f,a))))); \\ app_M & := & app] \end{array}$$

As for the representation of the axioms for combinators we have two choices. The first is to follow Martin-Löf and let equality in the model be intensional:

$$\begin{array}{rcl} [K_M axiom & : & (A,B:\texttt{Type}; a:M(A); b:M(B))I(app_M(app_M(K_M,a),b),a); \\ S_M axiom & : & (A,B,C:\texttt{Type}; g:M(A\dot{\to}B\dot{\to}C))); f:M(A\dot{\to}B); a:M(A)) \\ & & I(app_M(app_M(app_M(\texttt{S},g),f),a), app_M(app_M(g,a), app_M(f,a)))] \end{array}$$

As discussed above the intended model immediately satisfies these axioms. However, the term model does not because there we need to *reinterpret* equality as convertibility, and quotient formation is not a set forming operation in type theory.

So we need a second notion which includes an explicit uninterpreted equivalence relation as part of the structure:

$$\begin{array}{rcl}
[E & : & (A:\texttt{Type}; M(A); M(A))Set; \\
ref_E & : & (A:\texttt{Type}; a:M(A))E(a,a); \\
sym_E & : & (A:\texttt{Type}; a,b:M(A); E(a,b))E(b,a); \\
trans_E & : & (A:\texttt{Type}; a,a',a'':M(A); E(a,a'); E(a',a''))E(a,a''); \\
appcong & : & (A,B:\texttt{Type}; c,c':M(A\dot{\to}B); a,a':M(A); E(c,c'); E(a,a'))E(\texttt{app}(c,a),\texttt{app}(c',a')); \\
K_M axiom & : & (A,B:\texttt{Type}; a:M(A); b:M(B))E(app_M(app_M(K_M,a),b),a); \\
S_M axiom & : & (A,B,C:\texttt{Type}; g:M(A\dot{\to}B\dot{\to}C))); f:M(A\dot{\to}B); a:M(A)) \\
 & & E(app_M(app_M(app_M(\texttt{S},g),f),a), app_M(app_M(g,a),app_M(f,a)))]
\end{array}$$

We shall also need the notion of a *homomorphism* of typed combinatory algebras in the second sense. This a family of functions indexed by the types, which preserve the equivalence relation and the operations.

We can prove that the algebra of combinatory terms under convertibility is initial among typed combinatory algebras (in the second sense), since it has a unique homomorphism (up to extensional equality) to any other. The proof is the standard one interpreted in our constructive setting.

In the context of an arbitrary typed combinatory algebras (in the second sense), a homomorphism h is defined by structural recursion on combinatory terms:

$$\begin{array}{rcl}
h(\texttt{K}) & = & K_M \\
h(\texttt{S}) & = & S_M \\
h(\texttt{app}(c,a)) & = & app_M(h(c),h(a))
\end{array}$$

The proof that h really is a homomorphism is direct: each rule of conversion is mapped into an axiom for combinatory algebras; that h commutes with the operations is immediate from the definition.

The uniqueness proof is by induction on terms. Assume that h' is any homomorphism from the initial algebra, that is,

$$\begin{array}{l}
E(h'(\texttt{K}),K_M) \\
E(h'(\texttt{S}),S_M) \\
E(h'(\texttt{app}(c,a)),app_M(h'(c),h'(a)))
\end{array}$$

and it follows that h and h' are extensionally equal:

$$(A:\texttt{Type}; a,a':\texttt{T}(A); \texttt{I}(a,a'))E(h(a),h'(a'))$$

Furthermore, the model of glued values is given by the following substitution

$$[M := G; K_M := K_G; S_M := S_G; app_M := app_G; K_M axiom := K_G axiom; S_M axiom := S_G axiom]$$

where

$$\begin{array}{rcl}
G(A) & = & \Sigma p : [\![A]\!].Gl(p) \\
K_G & = & \langle [\![\mathtt{K}]\!], gl(\mathtt{K})\rangle \\
S_G & = & \langle [\![\mathtt{S}]\!], gl(\mathtt{S})\rangle \\
app_G(\langle q, y\rangle, \langle p, x\rangle) & = & app_{Gl}(q, y, p, x) \\
K_G axiom(\langle p, x\rangle, \langle q, y\rangle) & = & ref(\langle p, x\rangle) \\
S_G axiom(\langle p, x\rangle, \langle q, y\rangle, \langle r, z\rangle) & = & ref(\ldots)
\end{array}$$

which is a combinatory algebra in the first sense.

The quote homomorphism is just $fst \circ fst$. We need to prove

$$\begin{array}{l}
\mathtt{I}(\mathbf{q}(K_M), \mathtt{K}) \\
\mathtt{I}(\mathbf{q}(S_M), \mathtt{S}) \\
\mathtt{I}(\mathbf{q}(app_M(q, p)), \mathtt{app}(\mathbf{q}(q), \mathbf{q}(p)))
\end{array}$$

The first two are immediate and the third follows by definition of a glued value. Indeed, the glued values are precisely those for which the third conversion holds. So all the work is in showing that we have a typed combinatory algebra of glued values.

This use of initial algebra semantics is similar to its use for structuring compiler correctness proofs, see for example Thatcher, Wagner, and Wright [28].

3.1.6 Categorical glueing

In categorical glueing [16, 22] one starts with a functor $\mathtt{T} : \mathcal{C} \to \mathcal{S}$. The glueing (or sconing) construction is a new category, which has as objects arrows

$$\mathbf{q} : X \to \mathtt{T}(A)$$

of $\mathcal{S}$ and as arrows pairs $\langle c, f\rangle$, such that

$$\mathtt{T}(c) \circ \mathbf{q} = \mathbf{q}' \circ f.$$

The Freyd cover construction is a special case of this.

We now see the similarity between the interpretation of function spaces in our glued model and the interpretation of hom-sets in categorical glueing: the commuting square states a similar requirement to the requirement that quote commutes with application.

3.1.7 Optimization of a standard normalization proof

We shall compare our proof with Martin-Löf's proof of normalization for a version of intuitionistic type theory [20] (adapted to the the simple case of typed combinatory logic). This proof is also expressed as a model construction, but is standard in the sense that it defines a reducibility predicate à la Tait. Another proof of normalization for typed combinators (also using Tait-reducibility) was implemented in ALF by Gaspes and Smith [13], but the relationship between this proof and our extracted algorithm seems somewhat less direct.

We shall here show that we can optimize this proof too and again extract **nf**! When formalizing an informal proof one always has to make explicit certain choices which were left implicit. Here we have chosen a representation which makes the optimization process as simple as possible.

So Martin-Löf [20] defined the meaning [6] of a type A relative to a term a is a triple consisting of a normal term, a proof that it is reducible (which is a property of terms corresponding to the glueing property of values) and a proof that it is normalizable:

$$[\![\ ]\!] : (A : \mathtt{Type}; a : \mathtt{T}(A))Set$$

$$[\![A, a]\!] \quad = \quad \Sigma(Red(A), [ap]\mathtt{I}(a, \mathbf{q}(ap)))$$

The first two components are put together into a pair

$$\begin{array}{rcl} Red & : & (A : \mathtt{Type})Set \\ \mathbf{q} & : & (A : \mathtt{Type}; xp : Red(A))\mathtt{T}(A) \end{array}$$

$$\begin{array}{rcl} Red(A \dot{\to} B) & = & \Sigma(\mathtt{T}(A \dot{\to} B), [c]\Pi(Red(A), [ap]\Sigma(Red(B), [bp]\mathtt{I}(\mathtt{app}(c, \mathbf{q}(ap)), \mathbf{q}(bp))))) \\ \mathbf{q}_{A \dot{\to} B}(\langle c, f\rangle) & = & c \end{array}$$

The integrated normalization algorithm (proof) can now be defined by

$$[\![\ ]\!] : (A : \mathtt{Type}; a : \mathtt{T}(A))[\![A, a]\!]$$

$$\begin{array}{rcl} [\![\mathtt{K}]\!] & = & \langle\langle \mathtt{K}, \lambda([xp]\langle\langle \mathtt{app}(\mathtt{K}, \mathbf{q}(xp)), \lambda([yp]\langle xp, \mathtt{convK}\rangle)\rangle, \mathtt{convref}\rangle)\rangle, \mathtt{convref}\rangle \\ [\![\mathtt{S}]\!] & = & (large\ term) \\ [\![\mathtt{app}(f, a)]\!] & = & app_M([\![f]\!], [\![a]\!]) \end{array}$$

where we have used the following application operator in the model

$$app_M : (A, B : \mathtt{Type}; c : \mathtt{T}(A \dot{\to} B)); a : \mathtt{T}(A); p : [\![A \dot{\to} B, c]\!]; q : [\![A, a]\!])[\![B, \mathtt{app}(c, a)]\!]$$

$$app_M(p, q) \quad = \quad \langle fst(app_R(p, q)), \mathtt{convtrans}(\mathtt{convapp}(snd(app_R(p, q))))\rangle$$

where

$$app_R(p, q) = app(snd(fst(p)), fst(q))$$

The connection with the optimized algorithm should now be apparent: just remove all proof object for convertibility and simplify the types accordingly. This process could be made precise by using a suitable formalism for removing redundant information such as checked quantifiers or subsets. It would also be interesting to investigate automatic removal of redundant information along the lines of Takayama [26].

Berger [4] has provided a related analysis for a strong normalization proof of the typed λ-calculus, and shown that one gets the normalization algorithm of Berger and Schwichtenberg [5]. He uses an alternative framework and explains program extraction in terms of modified realizability. Only the predicate logic part of the proof, and not the parts involving induction, is treated explicitly.

[6] This is a model construction in a somewhat different sense than before, since $[\![\ ]\!]$ here is indexed by a term as well as a type.

3.2 Propositional calculus

3.2.1 Syntax

We now extend our object language with type formers corresponding to the logical constants to get full intuitionistic propositional calculus:

$$
\begin{array}{rcl}
\dot{\emptyset} & : & \mathtt{Type} \\
\dot{1} & : & \mathtt{Type} \\
\dot{\times} & : & (\mathtt{Type};\mathtt{Type})\mathtt{Type} \\
\dot{+} & : & (\mathtt{Type};\mathtt{Type})\mathtt{Type}
\end{array}
$$

There are also new term constructors corresponding to the introduction and elimination rules for propositional calculus [7]

$$
\begin{array}{rcl}
\mathtt{case0} & : & (C:\mathtt{Type})\mathtt{T}(\dot{\emptyset}\dot{\to}C) \\
\mathtt{<>} & : & \mathtt{T}(\dot{1}) \\
\mathtt{inl} & : & (A,B:\mathtt{Type};\mathtt{T}(A))\mathtt{T}(A\dot{+}B) \\
\mathtt{inr} & : & (A,B:\mathtt{Type};\mathtt{T}(B))\mathtt{T}(A\dot{+}B) \\
\mathtt{case} & : & (A,B,C:\mathtt{Type};\mathtt{T}(A\dot{\to}C);\mathtt{T}(B\dot{\to}C))\mathtt{T}(A\dot{+}B\dot{\to}C) \\
\mathtt{<\,,\,>} & : & (A,B:\mathtt{Type};\mathtt{T}(A);\mathtt{T}(B))\mathtt{T}(A\dot{\times}B) \\
\mathtt{fst} & : & (A,B:\mathtt{Type};\mathtt{T}(A\dot{\times}B))\mathtt{T}(A) \\
\mathtt{snd} & : & (A,B:\mathtt{Type};\mathtt{T}(A\dot{\times}B))\mathtt{T}(B)
\end{array}
$$

3.2.2 Intended semantics

The intended semantics is for types

$$
\begin{array}{rcl}
[\![\dot{\emptyset}]\!] & = & \emptyset \\
[\![\dot{1}]\!] & = & 1 \\
[\![A\dot{+}B]\!] & = & [\![A]\!]+[\![B]\!] \\
[\![A\dot{\times}B]\!] & = & [\![A]\!]\times[\![B]\!]
\end{array}
$$

and terms

$$
\begin{array}{rcl}
[\![\mathtt{case0}]\!] & = & \lambda([z]\mathit{case}_0(z)) \\
[\![\mathtt{<>}]\!] & = & \langle\rangle \\
[\![\mathtt{inl}(a)]\!] & = & \mathit{inl}([\![a]\!]) \\
[\![\mathtt{inr}(b)]\!] & = & \mathit{inr}([\![b]\!]) \\
[\![\mathtt{case}(d,e)]\!] & = & \lambda([z]\mathit{case}([x]\mathit{app}([\![d]\!],x),[y]\mathit{app}([\![e]\!],y),z)) \\
[\![\mathtt{<}a,b\mathtt{>}]\!] & = & \langle[\![a]\!],[\![b]\!]\rangle \\
[\![\mathtt{fst}(a)]\!] & = & \mathit{fst}([\![a]\!]) \\
[\![\mathtt{snd}(a)]\!] & = & \mathit{snd}([\![a]\!])
\end{array}
$$

[7] To make things simple we sometimes give inference rules instead of axioms; it would be easy to modify the approach to deal with a with axioms only.

From the intended semantics we immediately get a consistency proof

$$(a : \mathtt{T}(\dot{\emptyset}))\emptyset$$

which is nothing but the term interpretation function specialized to $\dot{\emptyset}$.

It is a matter of debate whether or not this can be seen as an internalization of the discussion in Martin-Löf [21] on simple-minded versus metamathematical consistency. On the one hand, it is quite tempting to look at the opposition object (syntactic) level versus meta (semantic) level (compare Tarski [27]) in the formalisation as the counterpart to the opposition syntax versus semantics according to the meaning explanations of Martin-Löf [21]. On the other hand, it can be argued that the real semantics contains completely different dimensions. For instance, the real semantics has to do with the way that we use the language, and this notion of 'use' is not captured by our formalisation.

We next extend the definition of conversion with the following rules

$$\begin{array}{l}
\mathtt{I}(\mathtt{app}(\mathtt{case}(d,e),\mathtt{inl}(a)),\mathtt{app}(d,a)) \\
\mathtt{I}(\mathtt{app}(\mathtt{case}(d,e),\mathtt{inr}(a)),\mathtt{app}(e,b)) \\
\mathtt{I}(\mathtt{fst}(\mathtt{<}a,b\mathtt{>}),a) \\
\mathtt{I}(\mathtt{snd}(\mathtt{<}a,b\mathtt{>}),b)
\end{array}$$

together with congruence rules for the new term constructors.

3.2.3 Normalization algorithm

We get an enriched model which can be used for normalization by interpreting the new type formers as in the intended semantics. We also extend the definition of quote:

$$\begin{array}{rcl}
\mathbf{q}_{\dot{1}}(\langle\rangle) & = & \mathtt{<>} \\
\mathbf{q}_{A \dot{+} B}(\mathit{inl}(p)) & = & \mathtt{inl}(\mathbf{q}_A(p)) \\
\mathbf{q}_{A \dot{+} B}(\mathit{inr}(q)) & = & \mathtt{inr}(\mathbf{q}_B(q)) \\
\mathbf{q}_{A \dot{\times} B}(\langle p,q\rangle) & = & \mathtt{<}\mathbf{q}_A(p),\mathbf{q}_B(q)\mathtt{>}
\end{array}$$

This is another application of pattern matching with dependent types, and here we really get a more compact definition. For example, there is no clause for $\dot{\emptyset}$, since there is no constructor for $\emptyset$. But again, it is easy to replace this by standard primitive recursive schemata for dependent types if one defines an auxiliary function for each of the syntactic type formers.

We also need to extend the interpretation function for terms in the enriched model (we omit cases which are the same as for the intended interpretation).

$$\begin{array}{rcl}
[\![\mathtt{case0}]\!] & = & \langle \mathtt{case0}, \lambda([z]\mathit{case}_0(z))\rangle \\
[\![\mathtt{case}(d,e)]\!] & = & \langle \mathtt{case}(\mathbf{q}([\![d]\!]),\mathbf{q}([\![e]\!])), \lambda([z]\mathit{case}([x]\mathit{app}_M([\![d]\!],x),[y]\mathit{app}_M([\![e]\!],y),z))\rangle
\end{array}$$

3.2.4 Normalization proof

The definition of Gl can also be defined by pattern matching:

$$Gl_{\dot{1}}(\langle\rangle) \;=\; 1$$

$$
\begin{array}{rcl}
Gl_{A\dot{+}B}(inl(p)) & = & Gl_A(p) \\
Gl_{A\dot{+}B}(inr(q)) & = & Gl_B(q) \\
Gl_{A\dot{\times}B}(\langle p,q\rangle) & = & Gl_A(p)\times Gl_B(q)
\end{array}
$$

and the normalization proof extends as well.

3.3 Inductively defined types

3.3.1 Syntax

We finally introduce natural numbers and Brouwer ordinals:

$$
\begin{array}{rcl}
\mathtt{N} & : & \mathtt{Type} \\
\dot{\mathcal{O}} & : & \mathtt{Type}
\end{array}
$$

The new constructors for terms are

$$
\begin{array}{rcl}
\mathtt{0} & : & \mathtt{T}(\mathtt{N}) \\
\mathtt{s} & : & (\mathtt{T}(\mathtt{N}))\mathtt{T}(\mathtt{N}) \\
\mathtt{rec} & : & (C:\mathtt{Type};\mathtt{T}(C);\mathtt{T}(\mathtt{N}\dot{\to}C\dot{\to}C))\mathtt{T}(\mathtt{N}\dot{\to}C) \\
\mathtt{0} & : & \mathtt{T}(\dot{\mathcal{O}}) \\
\mathtt{sup} & : & (\mathtt{T}(\mathtt{N}\dot{\to}\dot{\mathcal{O}}))\mathtt{T}(\dot{\mathcal{O}}) \\
\mathtt{ordrec} & : & (C:\mathtt{Type};\mathtt{T}(C);\mathtt{T}((\mathtt{N}\dot{\to}\dot{\mathcal{O}})\dot{\to}(\mathtt{N}\dot{\to}C)\dot{\to}C))\mathtt{T}(\dot{\mathcal{O}}\dot{\to}C)
\end{array}
$$

3.3.2 Intended semantics

$$
\begin{array}{rcl}
[\![\mathtt{N}]\!] & = & N \\
[\![\dot{\mathcal{O}}]\!] & = & \mathcal{O}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathtt{0}]\!] & = & 0 \\
[\![\mathtt{s}(a)]\!] & = & s([\![a]\!]) \\
[\![\mathtt{rec}(d,e)]\!] & = & \lambda([z]rec([\![d]\!],[x,y]app(app([\![e]\!],x),y),z)) \\
[\![\mathtt{0}]\!] & = & 0 \\
[\![\mathtt{sup}(b)]\!] & = & sup([x]app([\![b]\!],x)) \\
[\![\mathtt{ordrec}(d,e)]\!] & = & \lambda([z]ordrec([\![d]\!],[xf,yf]app(app([\![e]\!],\lambda(xf)),\lambda(yf)),z))
\end{array}
$$

The new conversion rules are

$$
\begin{array}{l}
\mathtt{I}(\mathtt{app}(\mathtt{rec}(d,e),\mathtt{0}),d) \\
\mathtt{I}(\mathtt{app}(\mathtt{rec}(d,e),\mathtt{s}(a)),\mathtt{app}(\mathtt{app}(e,a),\mathtt{app}(\mathtt{rec}(d,e),a))) \\
\mathtt{I}(\mathtt{app}(\mathtt{ordrec}(d,e),\mathtt{0}),d) \\
\mathtt{I}(\mathtt{app}(\mathtt{ordrec}(d,e),\mathtt{sup}(b)),\mathtt{app}(\mathtt{app}(e,b),\mathtt{ordrec}(d,e)\circ b))
\end{array}
$$

together with congruence rules for the new term constructors. Here we have used an auxiliary syntactic binary composition operator

$$
\circ:(A,B,C:\mathtt{Type};\mathtt{T}(B\dot{\to}C);\mathtt{T}(A\dot{\to}B))\mathtt{T}(A\dot{\to}C)
$$

$$c \circ b = \mathtt{app}(\mathtt{app}(\mathtt{S}, \mathtt{app}(\mathtt{K}, c)), b)$$

We note that it is sufficient to build the 'intended model' to prove equational consistency

$$(\mathtt{I}(\mathtt{0}, \mathtt{s}(\mathtt{0})))\emptyset$$

However, to prove for example that the constructor $\mathtt{s}$ is one-to-one for convertibility we need the normalization proof.

3.3.3 Normalization algorithm

The enriched model for inductively defined types is obtained by the same principle. Only constructors with functional arguments need to be reinterpreted, so natural numbers are interpreted as in the intended model, but ordinals are not:

$$\begin{array}{rcl} [\![\mathtt{N}]\!] & = & N \\ [\![\dot{\mathcal{O}}]\!] & = & \mathcal{O}_M \end{array}$$

where

$$\mathcal{O}_M : Set$$

has the following introduction rules:

$$\begin{array}{rcl} 0_M & : & \mathcal{O}_M \\ sup_M & : & (c : \mathtt{T}(\mathtt{N} \dot{\to} \dot{\mathcal{O}}); f : (N)\mathcal{O}_M)\mathcal{O}_M \end{array}$$

and the following recursion operator

$$ordrec_M : (C : Set; C; (\mathtt{T}(\mathtt{N} \dot{\to} \dot{\mathcal{O}}); (N)\mathcal{O}_M; (N)C)C; \mathcal{O}_M)C$$

The quote function has the following new clauses

$$\begin{array}{rcl} \mathbf{q}_{\mathtt{N}}(0) & = & \mathtt{0} \\ \mathbf{q}_{\mathtt{N}}(s(p)) & = & \mathtt{s}(\mathbf{q}_{\mathtt{N}}(p)) \\ \mathbf{q}_{\dot{\mathcal{O}}}(0_M) & = & \mathtt{0} \\ \mathbf{q}_{\dot{\mathcal{O}}}(sup_M(c, f)) & = & \mathtt{sup}(c) \end{array}$$

Term interpretation in the enriched model becomes

$$\begin{array}{rcl} [\![\mathtt{rec}(d, e)]\!] & = & \langle \mathtt{rec}(\mathbf{q}([\![d]\!]), \mathbf{q}([\![e]\!])), \lambda([z]rec([\![d]\!], [x, y]app_M(app_M([\![e]\!], x), y), z))\rangle \\ [\![\mathtt{0}]\!] & = & 0_M \\ [\![\mathtt{sup}(b)]\!] & = & sup_M(\mathbf{q}([\![b]\!]), [x]app_M([\![b]\!], x)) \\ [\![\mathtt{ordrec}(d, e)]\!] & = & \langle \mathtt{ordrec}(\mathbf{q}([\![d]\!]), \mathbf{q}([\![e]\!])), \\ & & \quad \lambda([z]ordrec_M([\![d]\!], \\ & & \qquad [a, xf, yf]app_M(app_M([\![e]\!], \langle a, \lambda(xf)\rangle), \langle \mathtt{ordrec}(\mathbf{q}([\![d]\!]), \mathbf{q}([\![e]\!])) \circ a, \lambda(yf)\rangle), \\ & & \qquad z))\rangle \end{array}$$

where we have omitted cases which are the same as for the intended interpretation.

3.3.4 Normalization proof

We extend the definition of reducible value:

$$\begin{array}{rcl} Gl_{\mathtt{N}}(p) & = & 1 \\ Gl_{\dot{\mathcal{O}}}(p) & = & Glord(p) \end{array}$$

where

$$Glord : (\mathcal{O}_M)Set$$

is inductively defined by the introduction rules

$$Glord(0_M)$$
$$(c : \mathtt{T}(\mathtt{N}\dot{\to}\dot{\mathcal{O}}); f : (N)\mathcal{O}_M; (p : N)Glord(f(p)) \times (\mathtt{I}(\mathtt{app}(c, \mathbf{q}(p)), \mathbf{q}((f(p))))))Glord(sup_M(c, f))$$

The normalization proof extends.

3.3.5 Proof that the constructors are one-to-one

We illustrate this point only in the case of natural numbers, though this method of proof works generally for any data type.

By definition of $\mathbf{q}()\mathtt{N}$, the constructor $\mathtt{s}$ commutes with the normalization function. Let us assume that $a, b : \mathtt{T}(\mathtt{N})$ satisfy $\mathtt{I}(\mathtt{s}(a), \mathtt{s}(b))$, we have then

$$I(\mathbf{nf}(\mathtt{s}(a)), \mathbf{nf}(\mathtt{s}(b)))$$

since conversion implies identity of normal forms, and then

$$I(\mathtt{s}(\mathbf{nf}(a)), \mathtt{s}(\mathbf{nf}(b)))$$

by commutation of **nf** and $\mathtt{s}$, and finally

$$I(\mathbf{nf}(a), \mathbf{nf}(b))$$

because $\mathtt{s}$ is one-to-one for the *intensional* equality; from this follows

$$\mathtt{I}(a, b)$$

because any term is convertible to its normal form.

4 Typed λ-calculus

We shall consider normalization to weak head normal form, that is, no normalization under λ is performed. The method is similar to the treatment of combinatory logic, and we present the normalization algorithms but not the proofs. These are done in an analogous way.

Instead we focus on the choice of syntax for the typed λ-calculus. Shall we use traditional named variables, de Bruijn indices, or perhaps some kind of categorical combinators; shall we use a presentation à la Curry or à la Church; shall we employ *explicit* or *implicit* substitutions, etc? We shall make two basic choices. Firstly, as for combinatory logic we shall use formulations à la Church, which arise by first considering formalization of natural deduction systems for intuitionistic implicational calculus. This is one approach; we do not claim that it is the canonical one. (Compare for example Altenkirch [2] and Huet [14] who use ordinary de Bruijn-indices in their implementations of proofs about λ-calculi.) Secondly, we focus on normal terms, since these are the only ones we need for building models. One model can then be used for normalization proofs of several essentially equivalent calculi. We illustrate this by building a model which can be used for the normalization proofs of both a weak λ-calculus with explicit substitutions (similar to the $\lambda\sigma$-calculus [1]) and a nameless version of the weak λ-calculus with implicit substitutions which was used by Martin-Löf [20].

4.1 Variables

We begin by defining sets of variables. These will then be used for formalizing sets of normal terms and certain sets of general terms. We will also introduce variable sequences, which correspond to renamings. We shall in particular define the identity renaming.

The basic point is that the rule for assuming a variable in typed λ-calculus corresponds to the logical rule of assumption

$$(\Gamma \dot{\ni} A)\mathtt{T}(\Gamma, A)$$

stating that A is true iff it is a member of the assumption list (*context*) Γ. Usually, the membership requirement is stated as a side-condition but here it is an assumption. If we define membership inductively

$$\dot{\ni} : (\Gamma : \mathtt{Context}; A : \mathtt{Type})Set$$

$$\begin{aligned} \mathtt{0} &: (\Gamma : \mathtt{Context}; A : \mathtt{Type})\Gamma\#A\dot{\ni}A \\ \mathtt{s} &: (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \Gamma\dot{\ni}A)\Gamma\#B\dot{\ni}A \end{aligned}$$

we get proof objects which correspond to bounded de Bruijn-indices, so we can think of $\Gamma\dot{\ni}A$ as the singleton set containing a variable of type A.

Contexts are defined by

$$\mathtt{Context} : Set$$

$$\begin{aligned} \mathtt{[]} &: \mathtt{Context} \\ . &: (\mathtt{Context}; \mathtt{Type})\mathtt{Context} \end{aligned}$$

We also introduce variable lists:

$$\dot{\supseteq} : (\mathtt{Context}; \mathtt{Context})Set$$

$$\begin{aligned} \mathtt{<>} &: (\Delta : \mathtt{Context})\Delta\dot{\supseteq}\mathtt{[]} \\ \mathtt{<,>} &: (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \Gamma : \Delta\dot{\supseteq}\Gamma; \Delta\dot{\ni}A)\Delta\dot{\supseteq}\Gamma.A \end{aligned}$$

The notation is suggested by the fact that $\Delta\dot{\supseteq}\Gamma$ iff $\Gamma\dot{\ni}A$ implies $\Delta\dot{\ni}A$ for all A, that is, Γ is a subset of Δ if both are considered as *sets* of assumptions.

Just as lists of terms represent substitutions, lists of variables represent reindexing (nameless renaming). Logically, they represent structural manipulations of the context by weakening, contraction, and exchange. We will use the identity reindexing defined by

$$\mathtt{id} : (\Gamma : \mathtt{Context})\Gamma\dot{\supseteq}\Gamma$$

$$\begin{array}{rcl} \mathtt{id}_{\mathtt{[]}} & = & \mathtt{<>} \\ \mathtt{id}_{\Gamma.A} & = & \mathtt{<s(id_\Gamma),0>} \end{array}$$

where

$$\mathtt{s} : (\Delta, \Gamma : \mathtt{Context}; B : \mathtt{Type}; \Delta \dot{\supseteq} \Gamma)\Delta.B \dot{\supseteq} \Gamma$$

$$\begin{array}{rcl} \mathtt{s(<>)} & = & \mathtt{<>} \\ \mathtt{s(<}vs, v\mathtt{>)} & = & \mathtt{<s(}vs\mathtt{),s(}v\mathtt{)>} \end{array}$$

is the weakening (lifting, projection) lifted to reindexings.

4.2 Normalization to weak head normal form

4.2.1 Normal terms

Weak head normal terms have either of the forms $f(a_1, \ldots, a_n)$ or $v(a_1, \ldots, a_n)$, where f is a λ-abstraction, v is a variable and $a_1, \ldots, a_n$ are weak head normal terms. Since f is a λ-abstraction it may contain an arbitrary (not necessary normal) term, and hence the definition of normal term `Nt` is parametric in the definition of general term `T`[8]. We define normal terms and lists of normal terms by a simultaneous inductive definition:

$$\begin{array}{rcl} \mathtt{Nt} & : & (\mathtt{Context}; \mathtt{Type})Set \\ \rightarrow_{\mathtt{Nt}} & : & (\mathtt{Context}; \mathtt{Context})Set \end{array}$$

$$\begin{array}{rcl} \mathtt{appf} & : & (\Delta, \Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \mathtt{T}(\Gamma.A, B); \Delta \rightarrow_{\mathtt{Nt}} \Gamma)\mathtt{Nt}(\Delta, A \dot{\rightarrow} B) \\ \mathtt{appv} & : & (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \Delta \dot{\ni} \Gamma \dot{\twoheadrightarrow} A; \Delta \rightarrow_{\mathtt{Nt}} \Gamma)\mathtt{Nt}(\Delta, A) \\ \mathtt{<>} & : & (\Delta : \mathtt{Context})\Delta \rightarrow_{\mathtt{Nt}} \mathtt{[]} \\ \mathtt{<,>} & : & (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \Delta \rightarrow_{\mathtt{Nt}} \Gamma; \mathtt{Nt}(\Delta, A))\Delta \rightarrow_{\mathtt{Nt}} \Gamma.A \end{array}$$

where $\dot{\twoheadrightarrow}$ is multivariate function space:

$$\dot{\twoheadrightarrow} : (\mathtt{Context}; \mathtt{Type})\mathtt{Type}$$

$$\begin{array}{rcl} (\Gamma.A) \dot{\twoheadrightarrow} B & = & \Gamma \dot{\twoheadrightarrow} (A \dot{\rightarrow} B) \\ \mathtt{[]} \dot{\twoheadrightarrow} B & = & B \end{array}$$

4.2.2 The glued model

We can now build our model. First we interpret types

$$[\![]\!] : (\mathtt{Context}; \mathtt{Type})Set$$

$$[\![A \dot{\rightarrow} B]\!]_\Gamma = \mathtt{Nt}(\Gamma, A \dot{\rightarrow} B) \times ([\![A]\!]_\Gamma \rightarrow [\![B]\!]_\Gamma)$$

[8] Alternatively, we could assume that λ-abstractions come normalized and replace `T` with `Nt` in the body of `appf`.

and the quote function

$$\mathbf{q} : (\Gamma : \mathtt{Context}; A : \mathtt{Type}; [\![A]\!]_\Gamma)\mathtt{Nt}(\Gamma, A)$$

$$\mathbf{q}_{A \dot{\to} B}(\langle c, f\rangle) = c$$

This can be lifted to yield interpretation of contexts and quotation of lists of values as well. We overload notation for these:

$$[\![\ ]\!] \ : (\mathtt{Context}; \mathtt{Context})Set$$

$$\mathbf{q}(:)(\Delta, \Gamma : \mathtt{Context}; [\![\Gamma]\!]_\Delta)\Delta \to_{\mathtt{Nt}} \Gamma$$

We use three auxiliary functions. The first is the projection function

$$\pi : (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \Gamma \dot{\ni} A; [\![\Gamma]\!]_\Delta)[\![A]\!]_\Delta$$

$$\begin{array}{rcl} \pi_{\mathtt{0}}(\langle ps, p\rangle) & = & p \\ \pi_{\mathtt{s}(v1)}(\langle ps, p\rangle) & = & \pi_{v1}(ps) \end{array}$$

The second is application in the model

$$app_M \ : (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; p : [\![A \dot{\to} B]\!]_\Gamma; q : [\![A]\!]_\Gamma)[\![B]\!]_\Gamma$$

$$app_M(\langle c, f\rangle, p) = app(f, p)$$

The third is identity in the model. We define it as the interpretation of the identity renaming. This is obtained by first interpreting variables and renamings in general:

$$[\![\ ]\!] : (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; v : \Delta \dot{\ni} \Gamma \dot{\twoheadrightarrow} A; ps : [\![\Gamma]\!]_\Delta)[\![A]\!]_\Delta$$

$$[\![v]\!]_{A \dot{\to} B}(ps) \quad = \quad \langle \mathtt{appv}(v, ' ps), \lambda([p][\![v]\!]_B(\langle ps, p\rangle))\rangle$$

This can be raised to an interpretation of lists of variables. Hence we get

$$id_M = [\![\mathtt{id}]\!](\langle\rangle)$$

4.2.3 Interpretation of a calculus of substitutions

As an example we shall prove normalization to whnf for a calculus with explicit substitutions similar to the $\lambda\sigma$-calculus. The syntax of terms is the following

$$\begin{array}{rcl} \mathtt{T} & : & (\Gamma : \mathtt{Context}; A : \mathtt{Type})Set \\ \longrightarrow & : & (\Delta, \Gamma : \mathtt{Context})Set \end{array}$$

$$\begin{array}{rcl} \mathtt{0} & : & (\Gamma : \mathtt{Context}; A : \mathtt{Type})\mathtt{T}(\Gamma.A, A) \\ \mathtt{s} & : & (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \mathtt{T}(\Gamma, B))\mathtt{T}(\Gamma.A, B) \\ \mathtt{app} & : & (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \mathtt{T}(\Gamma, A \dot{\to} B); \mathtt{T}(\Gamma, A))\mathtt{T}(\Gamma, B) \\ \dot{\lambda} & : & (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \mathtt{T}(\Gamma.A, B))\mathtt{T}(\Gamma, A \dot{\to} B) \\ \mathtt{[\ ]} & : & (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \mathtt{T}(\Gamma, A); \Delta \longrightarrow \Gamma)\mathtt{T}(\Delta, A) \end{array}$$

$$\begin{array}{rcl} \mathtt{id} & : & (\Gamma : \mathtt{Context})\Gamma \longrightarrow \Gamma \\ \mathtt{o} & : & (\Gamma, \Delta, \Theta : \mathtt{Context}; \Delta \longrightarrow \Theta; \Gamma \longrightarrow \Delta)\Gamma \longrightarrow \Theta \\ \mathtt{<>} & : & (\Gamma : \mathtt{Context})\Gamma \longrightarrow \mathtt{[]} \\ \mathtt{<,>} & : & (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \Delta \longrightarrow \Gamma; \mathtt{T}(\Delta, A))\Delta \longrightarrow \Gamma.A \end{array}$$

The interpretation of terms and term lists in the model is

$$\begin{array}{ll} [\![\ \ ]\!] : & (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \mathtt{T}(\Gamma, A); [\![\Gamma]\!]_\Delta)[\![A]\!]_\Delta \\ [\![\ \ ]\!] : & (\Theta, \Delta, \Gamma : \mathtt{Context}; \Delta \longrightarrow \Gamma; [\![\Delta]\!]_\Theta)[\![\Gamma]\!]_\Theta \end{array}$$

$$\begin{array}{rcl} [\![\mathtt{0}]\!]\langle ps, p\rangle & = & p \\ [\![\mathtt{s}(a)]\!]\langle ps, p\rangle & = & [\![a]\!](ps) \\ [\![\mathtt{app}(c, a)]\!](ps) & = & app_M([\![c]\!](ps), [\![a]\!](ps)) \\ [\![\dot{\lambda}(b)]\!](ps) & = & \langle \mathtt{appf}(b,' ps), \lambda([p][\![b]\!]\langle ps, p\rangle)\rangle \\ [\![a\,\mathtt{[}as\mathtt{]}]\!](ps) & = & [\![a]\!]([\![as]\!](ps)) \end{array}$$

$$\begin{array}{rcl} [\![\mathtt{id}]\!](ps) & = & ps \\ [\![cs \mathtt{o} bs]\!](ps) & = & [\![cs]\!]([\![bs]\!](ps)) \\ [\![\mathtt{<>}]\!](ps) & = & \langle\rangle \\ [\![\mathtt{<}as, a\mathtt{>}]\!](ps) & = & \langle [\![as]\!](ps), [\![a]\!](ps)\rangle \end{array}$$

The normal form function is then

$$\mathbf{nf}(a) = \mathbf{q}([\![a]\!](id_M))$$

We have the following conversion rules. (Leaving out congruence rules)

$$\begin{array}{lll} \mathtt{I}(\ ,\ ) & : & (\Gamma : \mathtt{Context}; A : \mathtt{Type}; \mathtt{T}(\Gamma, A); \mathtt{T}(\Gamma, A))Set \\ \mathtt{I}(\ ,\ ) & : & (\Delta, \Gamma : \mathtt{Context}; \Delta \longrightarrow \Gamma; \Delta \longrightarrow \Gamma)Set \end{array}$$

$$\begin{array}{l} \mathtt{I}(\mathtt{app}(\dot{\lambda}(b)\,\mathtt{[}as\mathtt{]}, a), b\,\mathtt{[<}as, a\mathtt{>]}) \\ \mathtt{I}(\mathtt{app}(c, a)\,\mathtt{[}as\mathtt{]}, \mathtt{app}(c\,\mathtt{[}as\mathtt{]}, c\,\mathtt{[}as\mathtt{]})) \\ \mathtt{I}(\mathtt{0}\,\mathtt{[<}as, a\mathtt{>]}, a) \\ \mathtt{I}(\mathtt{s}(b)\,\mathtt{[<}as, a\mathtt{>]}, b\,\mathtt{[}as\mathtt{]}) \\ \mathtt{I}(b\,\mathtt{[}bs\mathtt{]}\,\mathtt{[}as\mathtt{]}, b\,\mathtt{[}bs\mathtt{o}as\mathtt{]}) \end{array}$$

$$\begin{array}{l} \mathtt{I}(\mathtt{<>o}as, \mathtt{<>}) \\ \mathtt{I}(\mathtt{<}bs, b\mathtt{>o}as, \mathtt{<}bs\mathtt{o}as, b\,\mathtt{[}as\mathtt{]>}) \\ \mathtt{I}(\mathtt{ido}as, as) \\ \mathtt{I}((cs\mathtt{o}bs)\mathtt{o}as, cs\mathtt{o}(bs\mathtt{o}as)) \end{array}$$

We got these rules when trying to prove properties of the semantics, compare the methodology with Knuth-Bendix completion used by Curien [9] for turning the equations for cartesian closed categories into a term rewriting system.

It now follows[9]

$$(\Gamma : \mathtt{Context}; A : \mathtt{Type}; a, a' : \mathtt{T}(\Gamma, A); \mathtt{I}(a, a'))I([\![a]\!], [\![a']\!])$$

[9]The mechanical proofs have not yet been completed

and hence

$$(\Gamma : \texttt{Context}; A : \texttt{Type}; a, a' : \texttt{T}(\Gamma, A); \texttt{I}(a, a'))I(\mathbf{nf}(a), \mathbf{nf}(a'))$$

The normalization proof is to prove that

$$(\Gamma : \texttt{Context}; A : \texttt{Type}; a : \texttt{T}(\Gamma, A))\texttt{I}(a, \iota(\mathbf{nf}(a)))$$

where

$$\iota : (\Gamma : \texttt{Context}; A : \texttt{Type}; \texttt{Nt}(\Gamma, A))\texttt{T}(\Gamma, A)$$

is the injection of normal terms into general terms.

The proof of this is similar to the proof for combinatory logic, but in addition to a notion of glued value, we also need a notion of glued list of values. We omit the details of this construction.

4.2.4 Interpretation of the calculus of Martin-Löf 1973

Can we treat normalization to whnf for the traditional syntax? Yes, but we must remember that we must also change the implicitely defined substitution function so that it does not go under λ (otherwise we lose confluence, see Curien, Hardin, and Lèvy [10]). Therefore Martin-Löf [20] used a calculus where λ is replaced by `appf`:

$$\begin{array}{rcl}
\texttt{var} & : & (\Gamma : \texttt{Context}; A : \texttt{Type}; \Gamma \dot{\ni} A)\texttt{T}(\Gamma, A) \\
\texttt{appf} & : & (\Delta, \Gamma : \texttt{Context}; A, B : \texttt{Type}; \texttt{T}(\Gamma.A, B); \Delta \longrightarrow \Gamma)\texttt{T}(\Delta, A \dot{\to} B) \\
\texttt{app} & : & (\Gamma : \texttt{Context}; A, B : \texttt{Type}; \texttt{T}(\Gamma, A \dot{\to} B); \texttt{T}(\Gamma, A))\texttt{T}(\Gamma, B) \\
 & & \\
\texttt{<>} & : & (\Delta : \texttt{Context})\Delta \longrightarrow \texttt{[]} \\
\texttt{<,>} & : & (\Delta, \Gamma : \texttt{Context}; A : \texttt{Type}; \Gamma : \Delta \longrightarrow \Gamma; \texttt{T}(\Delta, A))\Delta \longrightarrow \Gamma.A
\end{array}$$

The interpretation functions for this calculus into the glued model can also easily be written (where clauses which are the same as for the calculus with explicit substitution above are omitted).

$$\begin{array}{rcl}
[\![\texttt{var}(v)]\!](ps) & = & \pi_v(ps) \\
[\![\texttt{appf}(b, as)]\!](ps) & = & \langle \texttt{appf}(b, \mathbf{q}([\![as]\!](ps))), \lambda([p][\![b]\!](\langle [\![as]\!](ps), p \rangle)) \rangle
\end{array}$$

The conversion rule is

$$\texttt{I}(\texttt{app}(\texttt{appf}(b, as), a), b\texttt{[<}as, a\texttt{>]})$$

where $\_\texttt{[\_]}$ and $\_ \texttt{o} \_$ are defined by a simultaneous recursive definition:

$$\begin{array}{rcl}
\texttt{var}(v)\texttt{[}as\texttt{]} & = & \pi_v(as) \\
\texttt{appf}(b, bs)\texttt{[}as\texttt{]} & = & \texttt{appf}(b, bs\texttt{o}as) \\
\texttt{app}(c, a)\texttt{[}as\texttt{]} & = & \texttt{app}(c\texttt{[}as\texttt{]}, a\texttt{[}as\texttt{]}) \\
 & & \\
\texttt{<>o}as & = & \texttt{<>} \\
\texttt{<}bs, b\texttt{>o}as & = & \texttt{<}bs\texttt{o}as, b\texttt{[}as\texttt{]>}
\end{array}$$

5 Related work

Catarina Coquand [6] has used a similar technique for full normalization with η-expansion in simply typed λ-calculus. She also proves a property of a normalization algorithm, which is composed by an interpretation function and a quote function. But we note the following essential differences

- A Kripke model is used instead of a glued model.
- Equality in the Kripke model is extensional, whereas it is sufficient to consider intensional equality between glued values.
- Her quote function is defined simultaneously with an auxiliary unquote function which is needed for interpreting variables. Both are defined by recursion on the type structure.

This algorithm is closely related to the normalization algorithm for full normalization with η-expansion in simply typed λ-calculus given by Berger and Schwichtenberg [5]. Their 'inversion of the evaluation functional' corresponds for example to the quote function from the Kripke model.

Categorical glueing was used by Lafont [15] for proving a coherence theorem for categorical combinators. However, he argued that the semantic component of his interpretation cannot directly be used for computing: 'Mais les *valeurs abstraites* de $A \rightarrow B$, avec leur composante fonctionelle, ne semblent guère "mechanisable" ' [15][page 18]. In contrast to this, we make use of the fact that these abstract values, when represented in our intuitionistic meta language, are indeed mechanizable. But, of course, the implementation of this meta language may still make use of an environment machine.

It is interesting to compare this situation with the following comments by Per Martin-Löf on a normalization result [18]: 'Of course, the fact that there is a not necessarily mechanical procedure for computing every function in the present theory of types does not require any proof at all for us, intelligent beings, who can understand the meaning of the types and the terms and recognize that the axioms and rules of inference of the theory are consonant with the intuitionistic notion of function according to which a function is the same as a rule or method.'

Related to this discussion is the following question: what kind of strategy (call-by-value, call-by-name, etc.) does the normalization algorithm extracted from these semantical arguments follow? The answer is simple: it is exactly the strategy used at the meta-level In a sense, the process of 'understanding' is represented by the computation of the semantics of a term.

Similar algorithms have also been considered in the context of metacircularity and partial evaluation. Pfenning and Lee [25] considered a notion of metacircularity for the polymorphic λ-calculus and defined an 'approximately metacircular interpreter' similar to our 'intended semantics'. Mogensen [23] considered similar notions for the untyped λ-calculus intended to be used as a foundation for partial evaluation. He defined a *self-interpreter* similar to our intended semantics and a *self-reducer* similar to our normalizer. Both these papers use *higher-order abstract syntax* for representing λ-terms, whereas we use a concrete representation.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *ACM Conference on Principles of Programming Languages, San Francisco*, 1990.

[2] T. Altenkirch. A formalisation of the strong normalization proof for System F in LEGO. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, Utrecht*, March 1993.

[3] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 199?

[4] U. Berger. Program extraction from normalization proofs. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, Utrecht*, March 1993. To appear.

[5] U. Berger and H. Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 203–211, July 1991.

[6] C. Coquand. Analysis of simply typed lambda-calculus in ALF. In *Draft Proceedings of the Winter Meeting, Tanum Strand*, January 1993.

[7] T. Coquand. Pattern matching with dependent types. In *Proceedings of The 1992 Workshop on Types for Proofs and Programs*, June 1992.

[8] T. Coquand and C. Paulin. Inductively defined types, preliminary version. In *LNCS 417, COLOG '88, International Conference on Computer Logic*. Springer-Verlag, 1990.

[9] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Pitman, 1986.

[10] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitution. Preliminary version, July 1991.

[11] P. Dybjer. Inductive families. *Formal Aspects of Computing*. To appear.

[12] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[13] V. Gaspes and J. M. Smith. Machine checked normalization proofs for typed combinator calculi. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, June 1992.

[14] G. Huet. Residual theory in λ-calculus: a complete Gallina development. 1993.

[15] Y. Lafont. *Logique, Categories & Machines. Implantation de Langages de Programmation guidee par la Logique Categorique*. PhD thesis, l'Universite Paris VII, January 1988.

[16] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[17] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[18] P. Martin-Löf. An intuitionistic theory of types. Unpublished report, 1972.

[19] P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109, 1975.

[20] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[21] P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[22] J. C. Mitchell and A. Scedrov. Notes on sconing and relators. 1992.

[23] T. Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.

[24] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an Introduction.* Oxford University Press, 1990.

[25] F. Pfenning and P. Lee. Metacircularity in the polymorphic lambda-calculus. *Theoretical Computer Science*, 89:137–159, 1991.

[26] Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. Submitted for publication in Journal of Symbolic Computation.

[27] A. Tarski. Der wahrheitsbegriff in den formalisierten sprachen. *Studia Philosophica*, 1:261–405, 1936.

[28] J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.

What is the status of pattern matching in type theory?

Thierry Coquand Jan M. Smith

Department of Computer Science, University of Göteborg/Chalmers

September 1993

In Martin-Löf's type theory sets are inductively defined by their introduction rules and induction on a set is expressed by an elimination rule, formulated in a way corresponding to Gentzen's eliminations rules in natural deduction. Coquand [2] has recently proposed pattern matching, that is case analysis together with recursion, as an alternative to elimination rules. The original motivations for this change of type theory were to get a simpler notation for proofs by induction and to make precise how implicitly defined constants can be added to type theory. But the addition of pattern matching to type theory also makes a reflection principle available which requires a universe in the usual formulation of type theory.

One can be reluctant to accept the addition of pattern matching to type theory, because it is a major change in the formalism which may have influences on the use as well as the understanding of type theory. We will in this note discuss pattern matching and its relation to the usual formulation of type theory.

The distinction between these two methods of proof is not restricted to type theory. Winskel's [8] rule induction and induction on derivations correspond to elimination rules and pattern matching, respectively, and Hallnäs [4] has also proposed a proof method similar to case analysis in the context of partial inductive definitions.

Some examples of pattern matching

When defining a function by recursion on a set in type theory, the elimination constant of the set must be used; for instance addition of two natural numbers, $add : (\mathsf{N};\mathsf{N})\mathsf{N}$, is introduced by

$$add(x,y) \;=\; \mathsf{natrec}(x,y,(x,z)\mathsf{succ}(z))\,.$$

With pattern matching, add would be directly introduced by the equations

$$\begin{cases} add(\mathsf{0},y) \;=\; y \\ add(\mathsf{succ}(x),y) \;=\; \mathsf{succ}(add(x,y))\,. \end{cases}$$

In the case of add, the definition by natrec is very close to the direct definition by the two equations, but if we instead look at subtraction $sub : (\mathsf{N};\mathsf{N})\mathsf{N}$, which by pattern matching can be defined by the equations

$$\begin{cases} sub(\mathsf{0},y) \;=\; \mathsf{0} \\ sub(\mathsf{succ}(x),\mathsf{0}) \;=\; \mathsf{succ}(x) \\ sub(\mathsf{succ}(x),\mathsf{succ}(y)) \;=\; sub(x,y)\,, \end{cases}$$

then a definition cannot be made by natrec which directly captures these equations; instead we are forced to use a higher order function which will give a definition which is harder to understand and which will result in more steps when computing $sub(m,n)$.

A function is introduced by pattern matching over a set by defining it for all possible constructors of the set; in particular, if a set has no constructor then a function can be introduced directly without any equations. Hence, since there is no introduction rule for $\mathsf{Id}(\mathsf{N},0,\mathsf{succ}(0))$, we immediately obtain a function

$$f(x) : \bot \ (x : \mathsf{Id}(\mathsf{N},0,\mathsf{succ}(0)))$$

and thereby a proof of Peano's fourth axiom $\neg\mathsf{Id}(\mathsf{N},0,\mathsf{succ}(0))$. Without pattern matching, a universe must be used to prove negated equalities [7].

A similar example of a propositions which cannot be proved without a universe but which is trivially true by pattern matching is obtained from

$$member : (a : A; l : \mathsf{List}(A))\ \mathsf{Set}$$

defined inductively by

$$\frac{}{member(a, a.l)} \qquad \frac{member(a,l)}{member(a, b.l)},$$

leaving out the proof-objects. Since there is no introduction rule for $member(a,\mathsf{nil})$, pattern matching directly gives a proof of $\neg member(a,\mathsf{nil})$. In order to prove this proposition without pattern matching we would first have to give another definition of the membership relation, this time by recursion using a universe:

$$\begin{cases} member'(a,\mathsf{nil}) & = \bot \\ member'(a,b.l) & = \mathsf{Id}(A,a,b) \lor member'(a,l)\,. \end{cases}$$

It is easy to show with elimination rules that $member(a,l)$ and $member'(a,l)$ are logically equivalent; hence $\neg member(a,\mathsf{nil})$ follows.

Another example along these lines is the following inductive definition of $\leq$ on the natural numbers:

$$\frac{}{0 \leq y} \qquad \frac{x \leq y}{\mathsf{succ}(x) \leq \mathsf{succ}(y)}\ .$$

A proof of transitivity of $\leq$ is straightforward by pattern matching but seems to require a universe when only using elimination rules.

In [3] a normalization proof by Tait's computability method for a combinator formulation of Gödel's system T is given in Martin-Löf's set theory both with elimination rules and with pattern matching. In this fairly large example it turns out that the length of the proof with pattern matching is about one third of the proof with elimination rules. One example in this proof where pattern matching makes it shorter is the definition of the computability predicate. This definition is intrinsically by recursion on the types but it can also be viewed as an inductive definition and, by using pattern matching, there are less steps in the proof than with the recursive definition. Since this inductive definition is not strictly positive, it is not even clear how an elimination rule should be formulated in this case.

A semantical analysis of simply typed λ-calculus, containing a normalization proof, is given in [1]. This analysis involves terms with explicit substitution and requires definition of sets by mutual induction; the proofs in [1] contain striking examples of simplifications obtained by the use of pattern matching. Although these proofs have not been carried out with elimination rules, we expect that the difference in length would be considerably greater than the one obtained in [3].

The above examples show that a proof by pattern matching can be much simpler than a proof by elimination rules; but there are even stronger examples: Hofmann has recently announced [5] a groupoid model for Martin-Löf's type theory with elimination rules in which pattern matching fails. An example which is false in the model but which can be proved by pattern matching is the following. Assume

$$h : (x : A)C(x, \mathsf{id}(x))$$

where $C(x, z) : \mathsf{Set}\ [x : A, z : \mathsf{Id}(A, x, x)]$. By pattern matching we can obtain an element f in

$$(x : A; z : \mathsf{Id}(A, x, x))C(x, z)$$

by simply defining f by

$$f(x, \mathsf{id}(x)) = h(x) : C(x, \mathsf{id}(x))\ [x : A]$$

because the only possible canonical form of $z : \mathsf{Id}(A, x, x)$ is $\mathsf{id}(x)$. In the groupoid model the interpretation of $\mathsf{Id}(A, a, a)$ may have more than one element and $(x : A; z : \mathsf{Id}(A, x, x))C(x, z)$ may be empty although $(x : A)C(x, \mathsf{id}(x))$ is inhabited; surprisingly, the interpretation of usual elimination rule for the Id sets can be justified in the model. However, if the set A is concretely given by an inductive definition, like the set of natural numbers, then it can be derived by elimination rules that $\mathsf{Id}(A, a, a)$ has only the element $\mathsf{id}(a)$ [5].

What are the conclusions of these examples?

As we have seen from the examples, pattern matching not only makes the proofs shorter and more readable, but it is also a proof method which entails more than the elimination rules for the sets we are reasoning about. The power comes from the possibility of excluding impossible cases in an inductive proof and this involves a kind of reflection on the inductive definition. Martin-Löf's motivation for the introduction of universes is that they express reflection principles; one can argue that pattern matching hides this reflection, instead it should be clearly seen in the rules one is using.

On the other hand, pattern matching is more flexible since one is not forced to add the full strength of an elimination rule when reasoning about an inductively defined set; for instance a case expression for the natural numbers can be directly introduced while with elimination rules one first have to formulate the stronger rules of natrec. Also, it seems that pattern matching reflects more closely the way induction proofs are done in ordinary mathematics than elimination rules do.

In [3] Martin-Löf has given a semantics for type theory by which the rules of type theory can be justified; this semantics also justifies pattern matching. But the semantics is not completely presented in [3]: judgements in contexts are explained by substituting arbitrary closed objects of appropriate types for the variables and it is crucial for the semantics what is meant by "substituting an arbitrary closed object." In [3] the interpretation was quite liberal so that $c(x) = d(x) : B\ [x : A]$ was interpreted as extensional equality which, in particular, is not decidable. Martin-Löf considers this extensional interpretation as a mistake and that instead the judgemental equality should be understood as definitional; then, obviously, there must be restrictions on what is meant by substituting an arbitrary closed object. It is, however, difficult to imagine a semantics along these lines which could justify Id-elimination without using case analysis and, hence, also would justify the last example above. A clearer understanding of the semantics of open objects may have consequences of the understanding of proofs by pattern matching as we can see from the following example.

In the above proof of $\neg\mathsf{Id}(\mathsf{N}, 0, \mathsf{succ}(0))$ by case analysis, the essential idea is that a case is ruled out because two terms beginning with different constructors cannot be definitional equal. A questionable use of case analysis along similar lines is the following: in Martin-Löf's type theory there is no reduction rule $\mathsf{if}\ (x, \mathsf{true}, \mathsf{false}) = x : \mathsf{Bool}\ [x : \mathsf{Bool}]$ so $\lambda x.x$ and $\lambda x.\mathsf{if}\ (x, \mathsf{true}, \mathsf{false})$ in $\mathsf{Bool} \to \mathsf{Bool}$ are not convertible; hence pattern matching will prove $\neg\mathsf{Id}(\mathsf{Bool}, \lambda x.x, \lambda x.\mathsf{if}\ (x, \mathsf{true}, \mathsf{false}))$ thereby ruling out the future addition of the conversion rule $\mathsf{if}\ (x, \mathsf{true}, \mathsf{false}) = x : \mathsf{Bool}\ [x : \mathsf{Bool}]$.

Acknowledgements. This note is a summary of discussions with many users of type theory. In particular, we would like to thank Michael Hedberg for several clarifying remarks.

References

[1] Catarina Coquand. A machine assisted semantical analysis of simply typed λ-calculus. Technical report, Dept. of Comp. Science, Chalmers Univ. of Technology, 1993.

[2] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[3] Veronica Gaspes and Jan M. Smith. Machine Checked Normalization Proofs for Typed Combinator Calculi. In *Proceeding from the logical framework workshop at Båstad, June 1992.*

[4] Lars Hallnäs. Partial Inductive Definitions. *Theoretical Computer Science*, (87), 1991.

[5] Martin Hofmann. A model of intensional martin-löf type theory in which unicity of identity proofs does not hold. Technical report, Dept. of Computer Science, University of Edinburgh, June 1993. Draft.

[6] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[7] Jan M. Smith. The Independence of Peano's Fourth Axiom from Martin-Löf's Type Theory without Universes. *Journal of Symbolic Logic*, 53(3), 1988.

[8] Glynn Winskel. *The Formal Semantics of Programming Languages. An Introduction.* The MIT Press.

On the Definition of the η-long Normal Form in Type Systems of the Cube

Gilles Dowek[*], Gérard Huet[†], Benjamin Werner[‡]
INRIA-Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

Introduction

In this paper we prove that the smallest transitive relation $<$ on normal terms such that

- if t is a strict subterm of u then $t < u$,
- if T is the normal form of the type of t and the term t is not a sort then $T < t$

is well-founded in the type systems of the cube [1]. This result is proved using the notion of *marked terms* introduced by de Vrijer [5]. One motivation for this theorem is to define the η-long form of a normal term in these type systems.

In simply typed λ-calculus, to define the η-long form of a normal term we first define the η-long form of a variable x of type $P_1 \to ... \to P_n \to P$ (P atomic) as the term $[y_1 : P_1]...[y_n : P_n](x\ y'_1\ ...\ y'_n)$ where y'_i is the η-long form of the variable y_i of type P_i. Then we define the η-long form of a normal term t well-typed of type $P_1 \to ... \to P_n \to P$ (P atomic) as

- if $t = [x : U]u$ then its η-long form is the term $[x : U]u'$ where u' is the η-long form of u,
- if $t = (x\ c_1\ ...\ c_p)$ then its η-long form is the term $[y_1 : P_1]...[y_n : P_n](x\ c'_1\ ...\ c'_p\ y'_1\ ...\ y'_n)$ where c'_i is the η-long form of the term c_i, and y'_i the η-long form of the variable y_i.

The definition of the η-long form of a variable is by induction over the structure of its type, and the definition of the η-long form of a normal term is by induction over the structure of the term itself. The η-long form appeared in [15] under the name of *long reduced form* and in [13] under the name of *η-normal form*, and was further investigated in [14], under the name of *extensional form*.

In systems with dependent types this definition is more complicated. First when $t = [x : U]u$ we have to take the η-long form of the term U too, then when $t = (x\ c_1\ ...\ c_p)$ we have to take the η-long form of the terms $P_1, ..., P_n$ too. So the well-foundedness of this definition is not so obvious, indeed P_i is not a subterm of t, but a subterm *of its type*. We prove the well-foundedness of this definition using the well-foundedness of the relation $<$.

Besides the definition of η-long form, this well-foundedness result has been used in [7] to prove the completeness of the resolution method in the systems of the cube and in [6] to prove the decidability of second order matching in these systems.

[*] Gilles.Dowek@inria.fr
[†] Gerard.Huet@inria.fr
[‡] Benjamin.Werner@inria.fr

1 The Cube of Typed λ-calculi

Definition: (Term)
The set of terms is inductively defined as

$$T \ ::= \ Prop \mid Type \mid x \mid (T\ T) \mid [x:T]T \mid (x:T)T$$

The symbols $Prop$ and $Type$ are called *sorts*, the terms x are called *variables*, the terms $(T\ T')$ *applications*, the terms $[x:T]T'$ *λ-abstractions* and the terms $(x:T)T'$ *products*. The notation $T \to T'$ is used for $(x:T)T'$ when x does not occur free in T'. In this paper we ignore variable renaming problems. A rigorous presentation would use de Bruijn indices. We consider that the application is left-associative and we write $(a\ b_1\ ...\ b_n)$ for $(\ ...\ (a\ b_1)\ ...\ b_n)$ when $n > 0$, and a when $n = 0$.

Notation: If t and u are two terms and x is a variable, we write $t[x \leftarrow u]$ for the term obtained by substituting u for x in t. We write $a =_\beta b$ when the terms a and b are β-convertible and $a =_{\beta\eta} b$ when the terms a and b are $\beta\eta$-convertible. We write $t \rhd u$ when t reduces (in one step) to u, $t \rhd^* u$ when t reduces in an arbitrary number of steps to u and $t \rhd^+ u$ when t reduces in at least one step to u.

Definition: (Context)
A *context* Γ is a list of pairs $< x, T >$ (written $x : T$) where x is a variable and T a term. The term T is called the *type* of x in Γ.
We write $[x_1 : T_1; ...; x_n : T_n]$ for the context with elements $x_1 : T_1, ..., x_n : T_n$ and $\Gamma_1\Gamma_2$ for the concatenation of the contexts Γ_1 and Γ_2.

Definition: (Typing rules)
We define inductively two judgements: Γ *is well-formed* and t *has type* T *in* Γ ($\Gamma \vdash t : T$) where Γ is a context and t and T are terms. These judgements are indexed by the parameter $\mathcal{R}$ which is a set of pairs of sorts that contains the pair $< Prop, Prop >$.

$$\frac{}{[\ ]\ \text{well-formed}}$$

$$\frac{\Gamma \vdash T : s \quad x \notin \Gamma}{\Gamma[x:T]\ \text{well-formed}}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma\ \text{well-formed}}{\Gamma \vdash Prop : Type}$$

$$\frac{\Gamma\ \text{well-formed} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma[x:T] \vdash U : s'}{\Gamma \vdash (x:T)U : s'} < s, s' > \in \mathcal{R}$$

$$\frac{\Gamma \vdash (x:T)U : s \quad \Gamma[x:T] \vdash t : U}{\Gamma \vdash [x:T]t : (x:T)U}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma \vdash t : (x:T)U \quad \Gamma \vdash u : T}{\Gamma \vdash (t\ u) : U[x \leftarrow u]}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s \quad T =_{\beta\eta} U}{\Gamma \vdash t : U} \; s \in \{Prop, Type\}$$

Remark: There are eight choices for the set $\mathcal{R}$ defining eight calculi. Examples of such calculi are the simply typed λ-calculus ($\mathcal{R} = \{< Prop, Prop >\}$), the $\lambda\Pi$-calculus [12] ($\mathcal{R} = \{< Prop, Prop >, < Prop, Type >\}$), the system F [11] ($\mathcal{R} = \{< Prop, Prop >, < Type, Prop >\}$), the system $F\omega$ [11] ($\mathcal{R} = \{< Prop, Prop >, < Type, Prop >, < Type, Type >\}$) and the Calculus of Constructions [2, 4] ($\mathcal{R} = \{< Prop, Prop >, < Prop, Type >, < Type, Prop >, < Type, Type >\}$).

Definition: (Well-typed term)
A term t is said to be *well-typed* in a context Γ iff there exists a term T such that $\Gamma \vdash t : T$.

Remark: The term $Type$ is not well-typed.

The following facts hold for every type system of the cube. The proofs are given in [8, 18]. We do not give them, as these results are not really new, and because this would be slightly redundant with respect to the next section.

Proposition: In all the systems of the cube each well-typed term has a unique type up to conversion.

Proposition: If $\Gamma[x : U] \vdash t : T$ and $\Gamma \vdash u : U$ then $\Gamma \vdash t[x \leftarrow u] : T[x \leftarrow u]$.

Proposition: Let Γ be a context and t and T be two terms such that $\Gamma \vdash t : T$. Then T is either the term $Type$ or a term well-typed in Γ, in this last case, the type of T is a sort.

Moreover if t is a variable, an application or an abstraction (but neither a sort nor a product) then $T \neq Type$.

Definition: (Atomic Term)
A term is said to be *atomic* if it has the form $(h\ c_1\ ...\ c_n)$ where h is a variable or a sort (in this last case with $n = 0$). The symbol h is called the *head* of this term.

Proposition: Let T be a well-typed normal term of type s for some sort s, T can be written in a unique way $T = (x_1 : P_1)...(x_n : P_n)P$ with P atomic. Moreover if $s = Type$ then $P = Prop$.

Proposition (Subject reduction): If $\Gamma \vdash t : T$ and $t \rhd^* u$ then $\Gamma \vdash u : T$.

Proposition (Normalization): In all the systems of the cube the $\beta\eta$-reduction relation is strongly normalizable.

Proposition (Confluence): In all the systems of the cube the $\beta\eta$-reduction relation is confluent.

Proposition (Church-Rosser): If $\Gamma \vdash t_1 : T$, $\Gamma \vdash t_2 : T$ and $t_1 =_{\beta\eta} t_2$, then there exists t such that $t_1 \rhd^* t$ and $t_2 \rhd^* t$ (and so $\Gamma \vdash t : T$).

Definition: (Subterm)
We consider well-typed normal terms labeled with the contexts in which they are well-typed: t_Γ. Let t_Γ such a term, we define by induction over the structure of t_Γ the set $Sub(t_\Gamma)$ of *strict subterms* of t_Γ:

- if t_Γ is a sort or a variable then $Sub(t_\Gamma) = \{\}$,

- if t_Γ is an application $t = (u\ v)$ then $Sub(t_\Gamma) = \{u_\Gamma, v_\Gamma\} \cup Sub(u_\Gamma) \cup Sub(v_\Gamma)$,
- if t_Γ is an abstraction $t = [x : P]u$ then $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$,
- if t_Γ is a product $t = (x : P)u$ then $Sub(t_\Gamma) = \{P_\Gamma, u_{\Gamma[x:P]}\} \cup Sub(P_\Gamma) \cup Sub(u_{\Gamma[x:P]})$.

Definition: (The Relation $<$)
Let $<$ be the smallest transitive relation defined on normal well-typed labelled terms such that

- if neither t nor u is a sort and t_Γ is a strict subterm of u_Δ then $t_\Gamma < u_\Delta$,
- if neither t nor T is a sort, and T_Γ is the (unique) normal form of the type of t_Γ in Γ then $T_\Gamma < t_\Gamma$.

2 Marked Terms

The main idea in this proof is to consider a new syntax for type systems such that the type of a well-typed term t is a subterm of t. So we define a syntax for type systems where each term is marked with its type. In fact, we only need to mark variables, applications and abstractions, but neither sorts nor products.

Definition: (Marked Terms)

$$T \ ::= \ Prop \mid Type \mid x^T \mid (T\ T)^T \mid ([x:T]T)^T \mid (x:T)T$$

Definition: Let t and u be marked terms and x a variable. We write $t[x \leftarrow u]$ for the term obtained by substituting u for x in t, remark that since the variable x may also occur in the marks we have to substitute both in the term and in the marks.

The notions of free variables and substitution straightforwardly extend to marked terms. Therefore β and η-reductions may be defined on marked terms in the same way they are on unmarked terms. We write $t \rhd u$ when t reduces (in one step) to u. Here also the contracted redex may be either in the term or in the marks. We write $t \rhd_\beta u$ if the contracted redex is a β-redex and $t \rhd_\eta u$ if the contracted redex is a η-redex.

We write R^* for the reflexive-transitive closure of a relation R and R^+ for its transitive closure (where R stands for $\rhd$, $\rhd_\beta$ or $\rhd_\eta$).

A marked term t is said to be normal (resp. β-normal, η-normal) if it contains no redex (resp. β-redex, η-redex).

Definition: (Contents of a marked term)
Let t be a marked term, the *contents* of t is the unmarked term $t^\#$ defined by induction over the structure of t:

- if t is a sort then $t^\# = t$,
- if $t = x^T$ then $t^\# = x$,
- if $t = (u\ v)^T$ then $t^\# = (u^\#\ v^\#)$,
- if $t = ([x : P]u)^T$ then $t^\# = [x : P^\#]u^\#$,

- if $t = (x : P)U$ then $t^{\#} = (x : P^{\#})U^{\#}$.

Remark: If t and u are marked terms and $t \rhd u$ then either $t^{\#} = u^{\#}$ (if the redex occurs in the marks) or $t^{\#} \rhd u^{\#}$.

Remark: If t is a marked term such that $t^{\#} \rhd_\beta u$ then there exists a marked term t' such that $t \rhd_\beta t'$ and $t'^{\#} = u$. Note that this proposition is false for η-reduction, indeed the term $t = [x : T](y^{([z:T]U\ x)}\ x)$ is not an η-redex, but $t^{\#} = [x : T](y\ x)$ is.

Definition: (Conversion on marked terms) Let t_1 and t_2 be two *marked* terms. We say that $t_1 =_{\beta\eta} t_2$ (respectively $t_1 =_\beta t_2$) if and only if $t_1^{\#} =_{\beta\eta} t_2^{\#}$ (respectively $t_1 =_\beta t_2$) [1].

Definition: A *marked context* is a list of pairs $< x, T >$ (written $x : T$) where x is a variable and T a marked term.

Definition: (Typing rules)
We define inductively two judgements: Γ *is well-formed* and t *has type* T *in* Γ ($\Gamma \vdash t : T$) where Γ is a marked context and t and T are marked terms.

$$\frac{}{[\ ]\text{ well-formed}}$$

$$\frac{\Gamma \vdash T : s \quad x \notin \Gamma}{\Gamma[x : T]\text{ well-formed}}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma\text{ well-formed}}{\Gamma \vdash Prop : Type}$$

$$\frac{\Gamma\text{ well-formed} \quad x : T \in \Gamma}{\Gamma \vdash x^T : T}$$

$$\frac{\Gamma \vdash T : s \quad \Gamma[x : T] \vdash U : s'}{\Gamma \vdash (x : T)U : s'} < s, s' > \in \boldsymbol{\mathcal{R}}$$

$$\frac{\Gamma \vdash (x : T)U : s \quad \Gamma[x : T] \vdash t : U}{\Gamma \vdash ([x : T]t)^{(x:T)U} : (x : T)U}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma \vdash t : (x : T)U \quad \Gamma \vdash u : T}{\Gamma \vdash (t\ u)^{U[x \leftarrow u]} : U[x \leftarrow u]}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s \quad T =_{\beta\eta} U}{\Gamma \vdash t : U}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma \vdash x^V : T \quad \Gamma \vdash W : s \quad V =_{\beta\eta} W}{\Gamma \vdash x^W : T}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma \vdash ([x : T]t)^V : U \quad \Gamma \vdash W : s \quad V =_{\beta\eta} W}{\Gamma \vdash ([x : T]t)^W : U}\ s \in \{Prop, Type\}$$

$$\frac{\Gamma \vdash (t\ u)^V : U \quad \Gamma \vdash W : s \quad V =_{\beta\eta} W}{\Gamma \vdash (t\ u)^W : U}\ s \in \{Prop, Type\}$$

[1]The convertibility relation used on marked terms is defined as $\beta\eta$-conversion on the underlying unmarked terms, whatever that marks are. We could chose to take the reflexive-symetric-transitive closure of the reduction relation instead, but this would imply to redo the proofs of propositions given in [8].

Remark that the three last rules permit to perform conversions in marks, in the same way as the rule above permit to perform conversion in the type of a term.

Definition: (Well-typed marked term)
A marked term t is said to be *well-typed* in a marked context Γ if and only if there exists a marked term T such that $\Gamma \vdash t : T$.

We now go to proving the basic properties of the system with marked terms. For several of the following properties, the proofs are very similar to what is done for the usual formulation using unmarked terms. In these cases, we do not detail the proof.

Definition: (Contents of a marked context)
Let $\Gamma = [x_1 : P_1; ...; x_n : P_n]$ be a marked context, the contents of Γ is the context

$$\Gamma^\# = [x_1 : P_1^\#; ...; x_n : P_n^\#]$$

Proposition: Let Γ be a marked context and t and T be marked terms such that $\Gamma \vdash t : T$. Then $\Gamma^\# \vdash t^\# : T^\#$.
Proof: By induction on the length of the derivation of $\Gamma \vdash t : T$.

Definition: (Atomic Term)
A marked term is said to be *atomic* if it has the form $(\; ... \; (h \; c_1)^{T_1} \; ... \; c_n)^{T_p}$ where h is a marked variable x^{T_0} or a sort s. The symbol h is called the *head* of this term.

Proposition (Uniqueness of Product Formation): If $(x : T_1)T_2 =_{\beta\eta} (x : U_1)U_2$ then $T_1 =_{\beta\eta} U_1$ and $T_2 =_{\beta\eta} U_2$. If s_1 and s_2 are two sorts such that $s_1 =_{\beta\eta} s_2$ then $s_1 = s_2$.
Proof: The proof considers the raw λ-terms obtained by replacing typed λ-abstraction $[x : T]t$ by $\lambda x.t$. On these raw terms, the Church-Rosser property holds. The result follows quite easily [8, 18].

Proposition (Substitution): If $\Gamma[x : U]\Gamma' \vdash t : T$ and $\Gamma \vdash u : U$ then $\Gamma\Gamma'[x \leftarrow u] \vdash t[x \leftarrow u] : T[x \leftarrow u]$.
Proof: By induction on the length of the derivation of $\Gamma[x : U] \vdash t : T$.

Proposition (Weakening): If $\Gamma\Gamma' \vdash t : T$ and $\Gamma \vdash U : s$ then $\Gamma[x : U]\Gamma' \vdash t : T$.
Proof: By induction on the length of the derivation of $\Gamma\Gamma' \vdash t : T$.

Proposition: If $\Gamma \vdash t : T$ then either $T = Type$ or there exists a sort s such that $\Gamma \vdash T : s$. Moreover if t is a variable, an application or an abstraction (but neither a sort nor a product) then $T \neq Type$.
Proof: By induction on the length of the derivation of $\Gamma \vdash t : T$. For induction loading one simultaneously proves that if $\Gamma = \Delta[z : V]\Delta'$, then $\Delta \vdash V : s$.

Proposition (Stripping):

- If $\Gamma \vdash Prop : T$ then $T = Type$,
- if $\Gamma \vdash x^T : U$ then there is some declaration $x : V$ in Γ such that $\Gamma \vdash T : s$ for some sort s and $T =_{\beta\eta} U =_{\beta\eta} V$,
- if $\Gamma \vdash (x : T_1)T_2 : U$ then there exists three sorts s, s_1, s_2 such that $U =_{\beta\eta} s$, $\Gamma \vdash T_1 : s_1$, $\Gamma[x : T_1] \vdash T_2 : s_2$ and $< s_1, s_2 > \in \boldsymbol{\mathcal{R}}$,

- if $\Gamma \vdash ([x:T]t)^U : V$ then there exists a term W and two sorts s_1 and s_2 such that $U =_{\beta\eta} V =_{\beta\eta} (x:T)W$, $\Gamma \vdash T : s_1$, $\Gamma[x:T] \vdash t : W$, $\Gamma[x:T] \vdash W : s_2$ and $< s_1, s_2 > \in \boldsymbol{\mathcal{R}}$,
- if $\Gamma \vdash (t\ u)^T : U$ then there exists two terms V and W such that $T =_{\beta\eta} U =_{\beta\eta} W[x \leftarrow u]$, $\Gamma \vdash t : (x:V)W$, $\Gamma \vdash u : V$.

Proof: By induction on the length of the derivation.

Corollary: In all the systems of the cube each well-typed term has a unique type up to conversion.

Corollary: Let t be a normal well-typed term, t is either an abstraction, a product or an atomic term.

Corollary: Let T be a well-typed normal marked term of type s for some sort s, T can be written in a unique way $T = (x_1 : P_1)...(x_n : P_n)P$ with P atomic. Moreover if $s = Type$ then $P = Prop$.

Corollary: Let Γ be a marked context and t and T be marked terms such that $\Gamma \vdash t : T$ and t is a variable, an application or an abstraction. Then T is convertible to the outermost mark of t. For instance if $t = (u\ v)^U$ then T is convertible to U.

Proposition (Convertibility of contexts): If $\Gamma[x:T]\Gamma'$ and $\Gamma[x:U]\Gamma'$ are well-formed contexts and $\Gamma \vdash T : s$, $\Gamma \vdash U : s$, $T =_{\beta\eta} U$ and $\Gamma[x:T]\Gamma' \vdash t : V$ hold then $\Gamma[x:U]\Gamma' \vdash t : V$
Proof: By induction on the length of the derivation of $\Gamma[x:T]\Gamma' \vdash t : V$.

Proposition (subject β-reduction): If $\Gamma \vdash t : T$ and $t \rhd_\beta t'$ then $\Gamma \vdash t' : T$.
Proof: By induction over the structure of t.

Proposition (Geuvers): If $\Gamma \vdash t : u$ and $t =_{\beta\eta} (x:T)U$ then $t \rhd_\beta (x:V)W$. If $\Gamma \vdash t : u$, given a sort s such that $t =_{\beta\eta} s$, one has $t \rhd_\beta s$.
Proof: The proof goes as in [8], using again raw λ-terms.

Proposition (Strengthening): If $\Gamma[x:U]\Gamma' \vdash t : T$ and x occurs free neither in Γ' nor in t then there exist a term T' such that $\Gamma\Gamma' \vdash t : T'$.
Proof: We may first remark that if T is some sort s, we may chose $T' = s$, as $T' =_{\beta\eta} s$, which implies $T' \rhd^*_\beta s$, and thus $\Gamma\Gamma' \vdash t : s$, by subject β-reduction and conversion. The proof then goes by induction over the structure of t:

- If $t = (t_1\ t_2)^A$, we know that $\Gamma[x:U]\Gamma' \vdash t_1 : (z:C)D$ and $\Gamma[x:U]\Gamma' \vdash t_2 : C$. The induction hypothesis ensures that $\Gamma\Gamma' \vdash t_1 : E$ and $\Gamma\Gamma' \vdash t_2 : C'$. As $E =_{\beta\eta} (z:C)D$, Geuvers' lemma implies that $E \rhd^*_\beta (z:C'')D'$. The conversion rule gives $\Gamma\Gamma' \vdash t_2 : C''$, and so $\Gamma\Gamma' \vdash (t_1\ t_2)^{D'[z \leftarrow t_2]} : D'[z \leftarrow t_2]$. The induction hypothesis also implies $\Gamma\Gamma' \vdash A : s$, and so we finally have $\Gamma\Gamma' \vdash (t_1\ t_2)^A : D'[z \leftarrow t_2]$.
- If $t = ([z:A]t_0)^B$, the stripping lemma ensures that $\Gamma[x:U]\Gamma'[z:A] \vdash t_0 : (z:A)C$, $\Gamma[x:U]\Gamma' \vdash A : s_1$, $\Gamma[z:U]\Gamma'[z:A] \vdash C : s_2$ and $< s_1, s_2 > \in \boldsymbol{\mathcal{R}}$. We may apply the induction hypothesis in order to get $\Gamma\Gamma'[z:A] \vdash t_0 : D$, $\Gamma\Gamma' \vdash A : s_1$, $\Gamma\Gamma'[z:A] \vdash C : s_2$. Applying Geuvers's lemma we get $D \rhd^*_\beta (z:A')C'$. Uniqueness of typing then ensures that $\Gamma\Gamma'[z:A] \vdash C' : s_2$, and therefore $\Gamma\Gamma' \vdash (z:A)C' : s_2$. This is sufficient to derive the judgement $\Gamma\Gamma' \vdash ([z:A]t_0)^{(z:A)C'} : (z:A)C'$ and finally $\Gamma\Gamma' \vdash ([z:A]t_0)^B : (z:A)C'$, as $\Gamma\Gamma' \vdash B : s$.

The other cases are straightforward.

Proposition (subject η-reduction): If $\Gamma \vdash t : T$ and $t \rhd_\eta t'$ then $\Gamma \vdash t' : T$.
Proof: By induction over the structure of t, all the cases are straightforward, but the one in which t is itself the reduced η-redex. In this case $t = ([x : U](u\ x^V)^W)^X$ and $t' = u$. Using twice the stripping lemma we get $\Gamma[x : U] \vdash u : (y : A)B$ with $(y : A)B =_{\beta\eta} T$. By the strengthening lemma we get $\Gamma \vdash u : C$ and by unicity of typing $C =_{\beta\eta} (y : A)B$. As $T \neq Type$ we have $\Gamma \vdash T : s$ thus by conversion $\Gamma \vdash u : T$.

We have remarked above that even if $t^{\#}$ is an η-redex then t is not necessarily an η-redex. We prove now that if we perform enough reductions in the marks, the term t becomes an η-redex. For instance the term $t = [x : T](y^{([z:T]U\ x)}\ x)$ reduces to $[x : T](y^U\ x)$ which is an η-redex. More precisely, we eventually want to prove the following: if t is $\beta\eta$-normal then $t^{\#}$ is $\beta\eta$-normal.

Proposition: If $\Gamma[x : U]\Gamma' \vdash t : T$, t is β-normal and x does not occur free neither in Γ' nor in $t^{\#}$ and in T, then x does not occur free in t.
Proof: By induction over the structure of t:

- If $t = (z : A)B$, we have $\Gamma[x : U]\Gamma' \vdash A : s$ which implies that x does not occur free in A. We then apply the induction hypothesis to $\Gamma[x : U]\Gamma'[z : A] \vdash B : s'$ which ensures that x does not occur free in B.
- It $t = x^A$, we just apply the induction hypothesis to $\Gamma[x : U]\Gamma' \vdash A : s$.
- If $t = ([z : B]C)^D$, the same reasoning as above ensures that x does not occur free in B and in D. We know that D (or T) is convertible to a product, and thus $D \rhd_\beta^* (z : B')E$. We may then apply the induction hypothesis to $\Gamma[x : U]\Gamma'[z : B] \vdash C : E$.
- If $t = (\ ...\ (z^{T_0}\ c_1)^{T_1}\ ...\ c_n)^{T_p}$, we first remark that x does not occur free in any T_i, as they are all well-typed of some sort in $\Gamma[x : U]\Gamma'$. The stripping lemma ensures that T_0 is convertible to some product, and thus $T_0 \rhd_\beta^* (y : A)B$. As $\Gamma[x : U]\Gamma' \vdash c_1 : A$, we know that x does not occur free in c_1. This implies that x does not occur free in $B[x \leftarrow c_1]$. We may then iterate that reasoning and prove by induction over i that x does not occur free in any c_i.

Proposition: Let t be a well-typed marked term. If t is $\beta\eta$-normal then $t^{\#}$ is $\beta\eta$-normal.
Proof: We already know that $t^{\#}$ is β-normal. Now suppose that t is not η-normal. So we have a subterm u of t such that $u^{\#}$ is an η-redex: $u = ([x : A](v\ x^B)^C)^D$ with x not occurring free in $v^{\#}$. It is easy to prove, by induction over the structure of v, that we may apply the previous proposition to every mark of v. This implies that x does not occur free in v which leads to a contradiction.

3 Normalization of Marked Terms

In this section we prove that each well-typed marked term has a unique normal form.

Proposition: Let Γ be a marked context and t a term well-typed in Γ. If the term t has the form x^T, $(u\ v)^T$, $([x : P]u)^T$ then the term T is well-typed in Γ and its type is a sort.
Proof: By induction on the length of the derivation of $\Gamma \vdash t : T$.

Definition: (Translation of a marked term into a unmarked term)

Let Γ be a marked context and t be a marked term well-typed in Γ or which is a sort. When t has the form x^T, $(u\ v)^T$, $([x:P]u)^T$ then let s be the type of T. If $s = Type$ then T has the form $T = (x_1 : A_1)...(x_n : A_n)Prop$ we let $\overline{T} = (x_1 : A_1)...(x_n : A_n)o^{Prop}$, otherwise we let $\overline{T} = T$.

We define by induction on t a term t°.

- if t is a sort we let $t^\circ = t$,
- if $t = x^T$ then we let $t^\circ = ([z : Prop]x\ \overline{T}^\circ)$,
- if $t = (u\ v)^T$ then we let $t^\circ = ([z : Prop](u^\circ\ v^\circ)\ \overline{T}^\circ)$,
- if $t = ([x : P]u)^T$ then we let $t^\circ = ([z : Prop][x : P^\circ]u^\circ\ \overline{T}^\circ)$,
- if $t = (x : P)Q$ then we let $t^\circ = (x : P^\circ)Q^\circ$.

This translation is similar to the ones defined in [12, 10].

Definition: (Translation of a marked context into a unmarked context)

Let $\Gamma = [x_1 : P_1; ...; x_n : P_n]$ be a marked context, we let $\Gamma^\circ = [x_1 : P_1^\circ; ...; x_n : P_n^\circ]$.

Proposition: $a^\circ[x \leftarrow b^\circ] \rhd^* (a[x \leftarrow b])^\circ$

Proof: By induction over the structure of a.

If $a = x^T$ then:

$$a^\circ = ([z : Prop]x\ \overline{T}^\circ)$$

$$a^\circ[x \leftarrow b^\circ] = ([z : Prop]b^\circ\ \overline{T}^\circ[x \leftarrow b^\circ]) \rhd^* b^\circ = (x^T[x \leftarrow b])^\circ = (a[x \leftarrow b])^\circ$$

The other cases are a simple application of the induction hypothesis. For instance if a is an application $a = (t\ u)^T$. We have

$$a^\circ = ([z : Prop](t^\circ\ u^\circ)\ \overline{T}^\circ)$$

So

$$a^\circ[x \leftarrow b^\circ] = ([z : Prop](t^\circ[x \leftarrow b^\circ]\ u^\circ[x \leftarrow b^\circ])\ \overline{T}^\circ[x \leftarrow b^\circ])$$

By induction hypothesis we have

$$t^\circ[x \leftarrow b^\circ] \rhd^* (t[x \leftarrow b])^\circ$$

$$u^\circ[x \leftarrow b^\circ] \rhd^* (u[x \leftarrow b])^\circ$$

$$\overline{T}^\circ[x \leftarrow b^\circ] \rhd^* (\overline{T}[x \leftarrow b])^\circ$$

So

$$\begin{aligned} a^\circ[x \leftarrow b^\circ] &\rhd^* ([z : Prop]((t[x \leftarrow b])^\circ\ (u[x \leftarrow b])^\circ)(\overline{T}[x \leftarrow b])^\circ) \\ &= ([z : Prop]((t[x \leftarrow b])^\circ\ (u[x \leftarrow b])^\circ)(\overline{T[x \leftarrow b]})^\circ) \\ &= ((t[x \leftarrow b]\ u[x \leftarrow b])^{T[x \leftarrow b]})^\circ = (a[x \leftarrow b])^\circ \end{aligned}$$

Proposition: If $a \rhd b$ then $a^\circ \rhd^+ b^\circ$.

Proof: If $a = (([x:P]t)^T u)^U$ and $b = t[x \leftarrow u]$,

$$a^\circ = ([z:Prop]([z':Prop][x:P^\circ]t^\circ\ \overline{T}^\circ\ u^\circ)\ \overline{U}^\circ)$$

$$b^\circ = (t[x \leftarrow u])^\circ$$

By reducing first three β-redexes in a° we get $t^\circ[x \leftarrow u^\circ]$. Thus

$$a^\circ \rhd^+ t^\circ[x \leftarrow u^\circ]$$

and

$$t^\circ[x \leftarrow u^\circ] \rhd^* (t[x \leftarrow u])^\circ = b^\circ$$

thus

$$a^\circ \rhd^+ b^\circ$$

If $a = ([x:P](t\ x^P)^T)^U$ and $b = t$,

$$a^\circ = ([z:Prop][x:P^\circ]([z':Prop](t^\circ\ ([z'':Prop]x\ \overline{P}^\circ))\ \overline{T}^\circ)\ \overline{U}^\circ)$$

$$b^\circ = t^\circ$$

We reduce three β-redexes and one η-redex in a° to get b°.

The same holds if we reduce a redex in a subterm.

Proposition: Let Γ be a marked context and t and T two marked terms such that $\Gamma \vdash t : T$ in some system of the cube then $[o:Prop]\Gamma^\circ \vdash t^\circ : T^\circ$ in the Calculus of Constructions.
Proof: By induction on the length of the derivation of $\Gamma \vdash t : T$.

Lemma: The reduction on marked terms is strongly normalizable.

If there was an infinite reduction issued from a marked term t, we could build one issued from the unmarked term t°, in contradiction with the fact that reduction is strongly normalizable on well-typed terms in the Calculus of Constructions.

Proposition: Let a and b two normal marked terms well-typed in the marked context Γ. If $a^\# = b^\#$ then $a = b$.
Proof: By induction over the structure of a. Since a and b have the same contents, they are either both sorts, both variables, both abstractions, both products or both applications.

If they are, for instance, both applications, $a = (t\ u)^T$, $b = (v\ w)^U$ then the marked terms t and v are well-typed and normal in Γ and have the same contents so they are equal, and symmetrically the marked terms u and w are equal. The marked terms T and U are well-typed and normal in Γ. Thus $T^\#$ and $U^\#$ are normal, and both are types of $a^\# = b^\#$ in $\Gamma^\#$. So $T^\# = U^\#$ and thus $T = U$. We then conclude that $a = b$.

The same holds if they are both sorts, variables, abstractions or products.

Proposition: Let a and b be two marked terms. If $a \rhd^* b$ then $a^\# \rhd^* b^\#$.
Proof: By induction on the length of the derivation of $a \rhd^* b$.

Proposition (Unicity of normal forms): Let Γ be a marked context and t, a, b be marked terms well-typed in Γ such that a and b are normal terms and t reduces (in an arbitrary number of steps) to a and b. Then $a = b$.
Proof: The marked terms a and b are well-typed in Γ and their contents is the normal form of $t^\#$.

Proposition (Confluence): Let $\Gamma \vdash t : T$, $t \rhd^* t_1$ and $t \rhd^* t_2$. Then there exists a term u such that $t_1 \rhd^* u$ and $t_2 \rhd^* u$.
Proof: Let u_1 and u_2 be the normal forms of t_1 and t_2. By definition, we have $t_1 \rhd^* u_1$ and $t_2 \rhd^* u_2$. The terms u_1 and u_2 are normal, $t \rhd^* u_1$ and $t \rhd^* u_2$, thus $u_1 = u_2$.

Proposition (Church-Rosser): Let $\Gamma \vdash t_1 : T$ and $\Gamma \vdash t_2 : T$ be two derivable judgements. If $t_1 =_{\beta\eta} t_2$, then there exits a term u such that $t_1 \rhd^* u$ and $t_2 \rhd^* u$.
Proof: By induction on the length of the derivation of $t_1 =_{\beta\eta} t_2$.

4 Well-foundedness of the relation $<$

We want to associate to each unmarked term a normal marked term. We could consider the normal form of the unmarked term and mark each subterm by its type, but we would need then to mark the new subterms introduced as marks and we would have to prove that this process terminates. It is actually simpler to build the marked term by induction on the length of the typing derivation of the term.

Definition: A marked context $\Gamma = [x_1 : P_1; ...; x_p : P_n]$ is said to be normal if every P_i is normal.

Proposition: Let Γ_1 and Γ_2 be two normal well-formed marked contexts such that $\Gamma_1^\# = \Gamma_2^\#$. Then $\Gamma_1 = \Gamma_2$.
Proof: By induction on the common length of Γ_1 and Γ_2.

Proposition: Let a and b two marked terms well-typed in the marked context Γ. If $a^\# =_{\beta\eta} b^\#$ then $a =_{\beta\eta} b$.
Proof: Let c be the normal form of a and d the normal form of b. The term $c^\#$ is the normal form of $a^\#$ and $d^\#$ is the normal form of $b^\#$. Since $a^\# =_{\beta\eta} b^\#$ we have $c^\# = d^\#$, and thus $c = d$. Therefore $a =_{\beta\eta} b$.

Definition: (Translation of an unmarked term into a marked term)
Let us consider an unmarked judgement $\Delta \vdash a : A$ or Δ *well-formed* which has a derivation. By induction on the length of this derivation, we build in the first case a normal marked context Δ^* and normal marked terms a^* and A^* such that $\Delta^{*\,\#} =_{\beta\eta} \Delta$, $a^{*\,\#} =_{\beta\eta} a$, $A^{*\,\#} =_{\beta\eta} A$ and the judgement $\Delta^* \vdash a^* : A^*$ is derivable and in the second a normal marked context Δ^* such that $\Delta^{*\,\#} =_{\beta\eta} \Delta$ and the judgement Δ^* *well-formed* is derivable.

- If the last rule of the derivation is

$$\frac{}{[\,]\ \text{well-formed}}$$

  we let $\Delta^* = [\,]$.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash T : s}{\Gamma[x : T]\ \text{well-formed}}$$

  then by induction hypothesis we have built Γ^* and T^*. We let $\Delta^* = \Gamma^*[x : T^*]$.

- If the last rule of the derivation is

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash Prop : Type}$$

then by induction hypothesis we have built Γ^*. We let $\Delta^* = \Gamma^*$, $a^* = Prop$ and $A^* = Type$.

- If the last rule of the derivation is

$$\frac{\Gamma \text{ well-formed} \quad x : T \in \Gamma}{\Gamma \vdash x : T}$$

then by induction hypothesis we have built Γ^*. We have $\Gamma^{*\,\#} =_{\beta\eta} \Gamma$, so Γ^* contains a declaration $x : P$ and $P^\# =_{\beta\eta} T$. We let $\Delta^* = \Gamma^*$, $a^* = x^P$ and $A^* = P$.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash T : s \quad \Gamma[x : T] \vdash U : s'}{\Gamma \vdash (x : T)U : s'}$$

then by induction hypothesis we have built Γ^*, T^*, $(\Gamma[x : T])^*$ and U^*. Since $(\Gamma[x : T])^{*\,\#} =_{\beta\eta} \Gamma[x : T]$, the context $(\Gamma[x : T])^*$ has the form $\Gamma'[x : P]$ with Γ' and P normal $\Gamma'^\# =_{\beta\eta} \Gamma$ and $P^\# =_{\beta\eta} T$. Then since Γ^* and Γ' are both normal well-formed and have the same contents we have $\Gamma^* = \Gamma'$ and since T^* and P are both normal, well-typed in Γ^* and have the same contents $T^* = P$. We let $\Delta^* = \Gamma^*$, $a^* = (x : T^*)U^*$ and $A^* = s'$.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash (x : T)U : s \quad \Gamma[x : T] \vdash t : U}{\Gamma \vdash [x : T]t : (x : T)U}$$

then by induction hypothesis we have built Γ^*, $((x : T)U)^*$, $(\Gamma[x : T])^*$, t^* and U^*. Since $((x : T)U)^*$ is normal and $((x : T)U)^{*\,\#} =_{\beta\eta} (x : T)U$ the term $((x : T)U)^*$ is a product $(x : P)Q$, P and Q are normal $P^\# =_{\beta\eta} T$ and $Q^\# =_{\beta\eta} U$. In the same way since $(\Gamma[x : T])^*$ is normal and $(\Gamma[x : T])^{*\,\#} =_{\beta\eta} \Gamma[x : T]$, the context $(\Gamma[x : T])^*$ has the form $\Gamma'[x : R]$ with Γ' and R normal $\Gamma'^\# =_{\beta\eta} \Gamma$ and $R^\# =_{\beta\eta} U$. Since Γ^* and Γ' are normal well-formed and have the same contents $\Gamma^* = \Gamma'$. Since P and R normal well-typed in Γ^* and have the same contents, $P = R$. Since Q and U^* are normal well-typed in $\Gamma^*[x : P]$ and have the same contents $Q = U^*$. We let $\Delta^* = \Gamma^*$, $a^* = ([x : P]t^*)^{(x:P)Q}$ and $A^* = (x : P)Q$.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash t : (x : T)U \quad \Gamma \vdash u : T}{\Gamma \vdash (t\ u) : U[x \leftarrow u]}$$

then by induction hypothesis we have built Γ_1^*, t^*, $((x : T)U)^*$, Γ_2^*, u^* and T^*. Since the term $((x : T)U)^*$ is normal and $((x : T)U)^{*\,\#} =_{\beta\eta} (x : T)U$, the term $((x : T)U)^*$ has the form $(x : P)Q$, P and Q are normal $P^\# =_{\beta\eta} T$ and $Q^\# = U$. Since Γ_1^* and Γ_2^* are normal, well-formed and have the same contents, $\Gamma_1^* = \Gamma_2^*$. Since P and T^* normal, well-typed in Γ_1^* and have the same contents, $T^* = P$. We have $\Gamma_1^* \vdash t^* : (x : T^*)U^*$ and $\Gamma_1^* \vdash u^* : T^*$, thus $\Gamma_1^* \vdash (t^*\ u^*)^{U^*[x \leftarrow u^*]} : U^*[x \leftarrow u^*]$. Let v be the normal form of $(t^*\ u^*)^{U^*[x \leftarrow u^*]}$ and V be the normal form of $U^*[x \leftarrow u^*]$. We have $\Gamma_1^* \vdash v : V$. We let $\Delta^* = \Gamma_1^*$, $a^* = v$ and $A^* = V$.

- If the last rule of the derivation is

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash U : s \quad T =_{\beta\eta} U}{\Gamma \vdash t : U}$$

  then by induction hypothesis we have built Γ_1^*, t^*, T^*, Γ_2^* and U^*. Since Γ_1^* and Γ_2^* are normal well-formed and have the same contents they are equal. Since T^* and U^* are normal, well-typed in Γ_1^* and have the same contents, they are equal. We let $\Delta^* = \Gamma_1^*$, $a^* = t^*$ and $A^* = U^*$.

Theorem: The relation $<$ is well-founded.
Proof: Let t and u be two unmarked normal terms which are not sorts and which are well-typed in two contexts Γ and Δ.

If t_Γ is a strict subterm of u_Δ then by induction on the structure of u, the term t^* is a strict subterm of u^*. If t is the normal form of the type of u then t^* is either the outermost mark of u^* or a sort, since t is not a sort, t^* is not a sort, so it is the outermost mark of u^*.

So if $t_\Gamma < u_\Delta$ then t^* is a strict subterm of u^*.

Assume there exist an infinite decreasing sequence of terms $t_1, ..., t_i, ...$. For each i, t_{i+1}^* is a strict subterm of t_i^*, thus the sequence $t_1^*, ..., t_i^*, ...$ is an infinite sequence of marked terms such that t_{i+1}^* is a strict subterm of t_i^* which is impossible.

Remark: The theorem above can also be proved for a Calculus of the Cube with β-conversion only.

5 Application to defining the η-long normal form

Definition: (Measure of a Term)

Let Γ be a context and t a normal term well-typed of type T in Γ. We define by induction over the order $<$, the *measure* $\mu(t)$ of t

- If t is a sort then $\mu(t) = 1$,
- If $t = x$ and T then $\mu(t) = \mu(T) + 1$,
- If $t = (u\ v)$ then $\mu(t) = \mu(u) + \mu(v) + \mu(T)$,
- If $t = [x : U]v$ then $\mu(t) = \mu(U) + \mu(v) + \mu(T)$,
- If $t = (x : U)V$ then $\mu(t) = \mu(U) + \mu(V)$.

Now we can give the following definition by induction over $\mu(t)$.

Definition: (η-long form)

Let Γ be a context and t a normal term well-typed in Γ and $(x_1 : P_1)...(x_n : P_n)P$ (P atomic) the normal form of its type. Then

- if $t = [x : U]u$ then the η-long form of t in Γ is the term $[x : U']u'$ where U' is the η-long form of U in Γ and u' the η-long form of u in $\Gamma[x : U]$,
- if $t = (x : U)V$ the η-long form of t in Γ is the term $(x : U')V'$ where U' is the η-long form of U in Γ and V' the η-long form of V in $\Gamma[x : U]$,

- if $t = (w\ c_1\ ...\ c_p)$ the η-long form of t in Γ is the term $[x_1 : P'_1]...[x_n : P'_n](w\ c'_1\ ...\ c'_p\ x'_1\ ...\ x'_n)$ where c'_i is the η-long form of c_i in Γ, P'_i the η-long form of P_i in $\Gamma[x_1 : P_1; ...; x_{i-1} : P_{i-1}]$ and x'_i the η-long form of x_i in $\Gamma[x_1 : P_1; ...; x_i : P_i]$.

Note that $\mu(x_i) = \mu(P_i) + 1 \leq \mu((x_1 : P_1)...(x_n : P_n)P) < \mu(t)$.

6 The Relation $<'$

Definition: (The Relation $<'$)
Let $<'$ be the smallest transitive relation defined on normal well-typed terms such that

- if neither t nor u is a sort and t_Γ is a strict subterm of u_Δ then $t_\Gamma <' u_\Delta$,
- if neither t nor T is a sort and T_Γ is the η-*long* form of the normal form of the type of t_Γ in Γ and the term t is not a sort then $T_\Gamma <' t_\Gamma$.

We prove that the relation $<'$ is also well-founded.

Definition: (Measure of a Marked Term)
Let t be a marked term, we define by induction over the structure of t, the *measure* $\mu(t)$ of t

- If t is a sort then $\mu(t) = 1$,
- If $t = x^T$ then $\mu(t) = \mu(T) + 1$,
- If $t = (u\ v)^T$ then $\mu(t) = \mu(u) + \mu(v) + \mu(T)$,
- If $t = ([x : U]v)^T$ then $\mu(t) = \mu(U) + \mu(v) + \mu(T)$,
- If $t = (x : U)V$ then $\mu(t) = \mu(U) + \mu(V)$.

Definition: (η-long Form of a Marked Term)
Let Γ be a marked context and t be a normal marked term well-typed in Γ with the type $(x_1 : P_1)...(x_n : P_n)P$ (P atomic). The *η-long form* of the marked term t is defined by induction on $\mu(t)$.

- If $t = ([x : U]u)^T$ then the η-long form of t is $([x : U']u')^{T'}$ where U' is the η-long form of U in Γ, T' the η-long form of T in Γ and u' the η-long form of u in $\Gamma[x : U]$,
- if $t = (x : U)V$ then the η-long form of t is $(x : U')V'$ where U' is the η-long form of U in Γ and V' the η-long form of V in $\Gamma[x : U]$,
- if $t = (\ ...\ (w^{T_0}\ c_1)^{T_1}\ ...\ c_p)^{T_p}$ then the η-long form of t is

$$[x_1 : P'_1]...[x_n : P'_n](\ ...\ ((\ ...\ (w^{T'_0}\ c'_1)^{T'_1}\ ...\ c'_p)^{T'_p}\ x'_1)^{V'_1}\ ...\ x'_n)^{V'_n}$$

where c'_i is the η-long form of c_i in Γ, T'_i the η-long form of T_i in Γ, P'_i the η-long form of P_i in $\Gamma[x_1 : P_1; ...; x_{i-1} : P_{i-1}]$, P' the η-long form of P in $\Gamma[x_1 : P_1; ...; x_n : P_n]$, x'_i the η-long form of $x_i^{P_i}$ in $\Gamma[x_1 : P_1; ...; x_i : P_i]$ and $V'_i = (x_{i+1} : P'_1)...(x_{i+1} : P'_n)P'$.

Definition: (Normal η-long Translation of an Unmarked Term)

Let t be a term well-typed in the context Γ, we define its *normal η-long translation* t^+ as the η-long form of its translation t^*.

Theorem: The relation $<'$ is well-founded.

Proof: As in the proof of the well-foundedness of the relation $<$ we first prove that if $t <' u$ then t^+ is a strict subterm of u^+ and then that there is no infinite decreasing sequence for the relation $<'$.

Conclusion

We have proved that the relation $<$ is well-founded in all the type systems of the cube. Thus $\beta\eta$-long forms exist in these type systems and moreover the relation $<'$ is well-founded too.

Acknowledgements

The authors thank Christine Paulin who suggested them the idea of the normalization proof for marked terms.

References

[1] H. Barendregt, Introduction to Generalized Type Systems, *Journal of Functional Programming*, 1, 2, 1991, pp. 125-154.

[2] Th. Coquand, Une Théorie des Constructions, *Thèse de troisième cycle*, Université de Paris VII, 1985.

[3] Th. Coquand, An Algorithm for Testing Conversion in Type Theory, *Logical Frameworks*, G. Huet and G. Plotkin (Eds.), Cambridge University Press, 1991.

[4] Th. Coquand, G. Huet, The Calculus of Constructions, *Information and Computation*, 76, 1988, pp. 95-120.

[5] R. de Vrijer, Big Trees in a λ-calculus with λ-expressions as types, λ-calculus and Computer Science Theory, Lecture Notes in Computer Science 37, 1975, pp. 252-271.

[6] G. Dowek, A Second Order Pattern Matching Algorithm in the Cube of Typed λ-Calculi, *Proceedings of Mathematical Foundation of Computer Science*, Lecture Notes in Computer Science, 520, 1991, pp. 151-160. Rapport de Recherche 1585, INRIA, 1992.

[7] G. Dowek, A Complete Proof Synthesis Method for the Cube of Type Systems, *Journal of Logic and Computation* (to appear).

[8] H. Geuvers, *Logics and Types Systems*, PhD Thesis, Catholic University of Nijmegen, 1993.

[9] H. Geuvers, The Church-Rosser Property for $\beta\eta$-reduction in Typed Lambda Calculi, *Proceedings of Logic in Computer Science*, 1992, pp. 453-460.

[10] H. Geuvers, M.J. Nederhof, A Modular Proof of Strong Normalization for the Calculus of Constructions, Journal of Functional Programming, I, 2, 1991, pp. 155-189.

[11] J.Y. Girard, Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur, *Thèse de Doctorat d'État*, Université de Paris VII, 1972.

[12] R. Harper, F. Honsell, G. Plotkin, A Framework for Defining Logics, *Proceedings of Logic in Computer Science*, 1987, pp. 194-204.

[13] G. Huet. A Unification Algorithm for Typed Lambda Calculus. *Theoretical Computer Science*, 1, 1, 1975 pp. 27–57.

[14] G. Huet. Résolution d'équations dans des langages d'ordre 1,2, ... ω. *Thèse d'Etat*, Université Paris VII, 1976.

[15] D.C. Jensen and T. Pietrzykowski. Mechanizing ω-order Type Theory Through Unification. *Theoretical Computer Science*, 3, 1976, pp. 123-171.

[16] Z. Luo, *An Extended Calculus of Constructions*, PhD Thesis, University of Edinburgh, 1990.

[17] A. Salvesen, The Church-Rosser Theorem for LF with β/η-reduction, Manuscript, University of Edinburgh, 1989.

[18] B. Werner, Une Théorie des Constructions Inductives, *Thèse de Doctorat*, Université Paris VII, 1993, forthcoming.

Conservativity between logics and typed λ calculi

Herman Geuvers*
Faculty of Mathematics and Computer Science
University of Nijmegen, Netherlands

1 Introduction

When looking at systems of typed λ calculus from a logical point of view, there are some interesting questions that arise. One of them is whether the the formulas-as-types embedding from the logic into the typed λ calculus is complete, that is, whether the types that are inhabited in the typed λ calculus are provable (as formulas) in the logic. It is well-known that this is not a vacuous question: the 'standard' formulas-as-types embedding from higher order predicate logic into the Calculus of Constructions is not complete. (See [Berardi 1989], [Geuvers 1989] or [Geuvers 1993].) Another interesting issue is whether the typed λ calculus approach can help to solve questions about the logics or vice versa. An example of such a fruitful interaction is the proof of (strong) normalization for the Calculus of Constructions, which has as corollary in higher order predicate logic that cut elimination terminates. In this paper we want to treat questions of conservativity between systems of typed λ calculi (and hence between the logical systems that correspond with them according to the formulas-as-types embedding). On the one hand this is an issue of interest for the typed λ calculi themselves (can new type forming operators create inhabitants of previously empty types?) On the other hand, however, this is a nice example of how the formulas-as-types embedding can help to solve questions about logics by making use of typed λ calculi and vice versa.

If one sees a typed λ calculus as a logical system, one takes one specific universe (sort in the terminology of Pure Type Systems) to be interpreted as the universe of all formulas. Let's call this universe Prop. Now suppose that S_1 is a system of typed λ calculus containing the universe Prop, and suppose that S_2 is a system that extends S_1.

Definition 1.1 *The type system S_2 is a* conservative extension *of S_1 if for every context Γ and type A one has*

$$\left.\begin{array}{r}\Gamma \vdash_{S_1} A : \mathsf{Prop} \\ \Gamma \vdash_{S_2} M : A\end{array}\right\} \Rightarrow \exists N[\Gamma \vdash_{S_1} N : A].$$

The 'logical' intuition should be clear: if the formula A, taken from the smaller system, is provable in the larger system, then it is already provable in the smaller system. To make the connection with logics a bit more precise we recall that, if $L_1$and L_2 are logics and L_2 extends L_1, then L_2 is a *conservative extension* of L_1 if for all formulas φ and sets of fomulas Δ one has

$$\left.\begin{array}{r}\Delta \cup \{\varphi\} \text{ is a set of formulas of } L_1 \\ \Delta \vdash_{L_2} \varphi\end{array}\right\} \Rightarrow \Delta \vdash_{L_1} \varphi.$$

*e-mail: herman@cs.kun.nl

Now, let H be the formulas-as-types embedding from L_1 into S_1 and from L_2 into S_2. This means that for every finite set of formulas Δ' in L_i there is a specific context $\Gamma_{\Delta'}$ in S_i, in which all the declarations are made that are necessary for forming the types $H(\psi)$ (for $\psi \in \Delta$) in S_i. Furthermore this embedding H is *sound*:

$$\Delta \vdash_{L_i} \varphi \Rightarrow \exists N[\Gamma_{\Delta \cup \{\varphi\}}, \vec{p}{:}H(\Delta) \vdash_{S_i} N : H(\varphi)].$$

Here the $\vec{p}{:}H(\Delta)$ denotes a vector of variable-declarations $p_i : H(\psi_i)$ for each $\psi_i \in \Delta$. Hence the Δ is taken to be finite, which is not a real restriction. In the formulas-as-types embedding, the term N of type $H(\varphi)$ is defined by induction on the derivation of $\Delta \vdash \varphi$, so the formulas-as-types embedding not only maps formulas to types, but also derivations to proof-terms.

The formulas-as-types embedding is not always *complete*, where completeness means (in the terminology above) that for each formula φ and finite set of formulas Δ, taken from L_i one has

$$\Gamma_{\Delta \cup \{\varphi\}}, \vec{p}{:}H(\Delta) \vdash_{S_i} N : H(\varphi) \Rightarrow \Delta \vdash_{L_i} \varphi.$$

Some well-known examples of the formulas-as-types embedding are not complete, like the embedding of higher order predicate logic into the Calculus of Constructions. In this paper we are more interested in embeddings that are complete, in which case we usually speak of a *formulas-as-types isomorphism*. This is because of the following.

Proposition 1.2 *If the formulas-as-types embedding H is complete, then*

$$S_2 \textit{ is a conservative extension of } S_1 \Leftrightarrow L_2 \textit{ is a conservative extension of } L_1.$$

This proposistion can be useful in two ways, both of which will be applied in this paper. Note therefore that in the definition of conservativity (Definition 1.1) there is no requirement about a function that takes an inhabitant $M : \varphi$ in the larger system and returns an inhabitant $N : \varphi$ in the smaller system. However, if there is such a function, then the conservativity result will usually be much easier to prove for the typed λ calculi, because one just has to define the function and to show (by induction on the derivation or by induction on the structure of the term) that it preserves derivability. This is a purely *syntactic* conservativity proof. If it is not clear how such a function should be defined, it is better to look at the logics, in which case one can forget about functions all together and just look at provability. In this latter case a semantic approach suits very well to prove conservativity.

Here we study the conservativity relations inside the cube of typed λ calculi, a collection of eight type systems defined by Barendregt (see [Barendregt 1992]) to give a fine structure for the Calculus of Constructions. There is a close connection between the cube of typed λ calculi and a cube of logical systems, due to the formulas-as-types embedding from the latter into the first. To make this embedding more readily understandable it is often described in two steps, first from the logic to a typed λ calculus that is in direct correspondence with the logic and then from this latter typed λ calculus to a type system of the cube. (See [Barendregt 1992] but also [Geuvers 1993].) To strip our discussions about typed λ calculi from the need to first having to justify all kinds of meta theoretic reasoning, we work in the framework of 'Pure Type Systems'. (See [Geuvers and Nederhof 1991], [Barendregt 1992] or [Geuvers 1993].) This gives a general method for describing typed λ calculi. Moreover we can use all the well-known meta-theory for Pure Type Systems (PTSs).

The main result in this paper is that, if S_1 is a system in the cube that contains the system S_1, then S_2 is a conservative extension of S_1, unless S_2 is the Calculus of Constructions (CC) and S_1 is the second order dependent typed λ calculus λP2. But more interesting than this general result is maybe the proof, which is devided in four cases. The first case is to show that CC is not conservative over λP2. The second case is to show that the extension of a system by adding type dependency is conservative. The third is to show that an extension of a first order system (i.e. a system in the bottom plane of the cube) is always conservative. This leaves over one special case, which is to show that $\lambda\omega$ is not conservative over the polymorphic λ calculus, $\lambda 2$. The second and third case are by defining a mapping from terms in the larger system to the terms in the smaller system, which gives us a purely syntactic conservativity proof. The fourth case is more difficult, because it is not clear how to do this proof by purely syntactic means. We therefore define a semantics for the logical systems of higher and second order propositional logic (which are isomorphic to $\lambda\omega$ and $\lambda 2$ under by the formulas-as-types embedding) and show the conservativity on the level of the logics.

2 A fine structure for the Calculus of Constructions

2.1 Pure Type Systems

Our studies of the Calculus of Constructions and its subsystems will be done in the framework of 'Pure Type Systems'. This provides a generic way of describing systems of typed λ calculus. In fact one can only describe systems that have as type forming operator just the Π and as reduction rule just β. As the Calculus of Constructions is such a system, the Pure Type Systems (or PTSs) is the right framework for us.

The Pure Type Systems are formal systems for deriving judgements of the form

$$\Gamma \vdash M : A,$$

where both M and A are in the set of so called *pseudoterms*, a set of expressions from which the derivation rules select the ones that are typable. The Γ is a finite sequence of *declarations*, statements of the form $x : B$, where x is a variable and B is a pseudoterm. The idea is that a term M can only be of type A ($M : A$) relative to a typing of the free variables that occur in M and A. Before giving the precise definition of Pure Type Systems we define the set of pseudoterms T over a base set $\mathcal{S}$. (The dependency of T on $\mathcal{S}$ is usually ignored.)

Definition 2.1 *For $\mathcal{S}$ some set, the set of pseudoterms over $\mathcal{S}$, T, is defined by*

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{Var} \mid (\Pi\mathsf{Var}{:}\mathsf{T}.\mathsf{T}) \mid (\lambda\mathsf{Var}{:}\mathsf{T}.\mathsf{T}) \mid \mathsf{T}\mathsf{T},$$

where Var *is a countable set of expressions, called variables. Both Π and λ bind variables and hence we have the usual notions of* free variable *and* bound variable. *We adopt the λ-calculus notation of writing $FV(M)$ for the set of free variables in the pseudoterm M.*
On T *we have the usual notion of β-reduction, generated from*

$$(\lambda x{:}A.M)P \longrightarrow_\beta M[P/x],$$

where $M[P/x]$ denotes the substitution of P for x in M (done with the usual care to avoid capturing of free variables), and compatible with application, λ-abstraction and Π-abstraction. We also adopt from the untyped λ calculus the conventions of denoting the transitive reflexive closure of $\longrightarrow_\beta$ by $\twoheadrightarrow_\beta$ and the transitive symmetric closure of $\twoheadrightarrow_\beta$ by $=_\beta$.

The typing of terms is done under the assumption of specific types for the free variables that occur in the term.

Definition 2.2 *1. A* declaration *is a statement of the form $x : A$, where x is a variable and A a pseudoterm,*

2. A pseudocontext *is a finite sequence of declarations such that, if $x : A$ and $y : B$ are different declarations of the same pseudocontext, then $x \not\equiv y$,*

3. If $\Gamma = x_1{:}A_1, \ldots, x_n{:}A_n$ is a pseudocontext, the domain of Γ, $\mathsf{dom}(\Gamma)$ *is the set $\{x_1, \ldots, x_n\}$; for $x_i \in \mathsf{dom}(\Gamma)$.*

4. For Γ a pseudocontext, a variable y is Γ-fresh *(or just* fresh *if it is clear which Γ we are talking about) if $y \notin \mathsf{dom}(\Gamma)$.*

5. For Γ and Γ' pseudocontexts, $\Gamma' \setminus \Gamma$ is the pseudocontext which is obtained by removing from Γ' all declarations $x : A$ for which $x \in \mathsf{dom}(\Gamma)$.

Definition 2.3 *A* Pure Type System *(*PTS*) is given by a set $\mathcal{S}$, a set $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ and a set $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The* PTS *that is given by $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$ is denoted by $\lambda(\mathcal{S}, \mathcal{A}, \mathcal{R})$ and is the typed lambda calculus with the following deduction rules.*

$$(sort) \quad \vdash s_1 : s_2 \qquad \textit{if } (s_1, s_2) \in \mathcal{A}$$

$$(var) \quad \frac{\Gamma \vdash A : s}{\Gamma, x{:}A \vdash x : A} \qquad \textit{if } x \textit{ is } \Gamma\textit{-fresh}$$

$$(weak) \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash M : C}{\Gamma, x{:}A \vdash M : C} \qquad \textit{if } x \textit{ is } \Gamma\textit{-fresh}$$

$$(\Pi) \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3} \qquad \textit{if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$(\lambda) \quad \frac{\Gamma, x{:}A \vdash M : B \quad \Gamma \vdash \Pi x{:}A.B : s}{\Gamma \vdash \lambda x{:}A.M : \Pi x{:}A.B}$$

$$(app) \quad \frac{\Gamma \vdash M : \Pi x{:}A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

$$(conv) \quad \frac{\Gamma \vdash M : A \quad \Gamma \vdash B : s}{\Gamma \vdash M : B} \qquad A =_\beta B$$

If $s_2 \equiv s_3$ in a triple $(s_1, s_2, s_3) \in \mathcal{R}$, we write $(s_1, s_2) \in \mathcal{R}$. The equality in the conversion rule (conv) is the β-equality on the set of pseudoterms T.
The elements of $\mathcal{S}$ are called sorts, *the elements of $\mathcal{A}$ (usually written as $s_1 : s_2$) are called* axioms *and the elements of $\mathcal{R}$ are called* rules.

This is not the place to go into a detailed treatment of the meta-theoretic properties of PTSs. For details we refer to [Geuvers and Nederhof 1991], [Barendregt 1992] or [Geuvers 1993]. We only give the most important properties without proof.

Proposition 2.4 *In an arbitrary* $\mathsf{PTS}\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$*, the following holds.*

- *Substitution*
  If $\Gamma_1, x{:}A, \Gamma_2 \vdash M : B$ *and* $\Gamma_1 \vdash N : A$*, then* $\Gamma_1, \Gamma_2[N/x] \vdash M[N/x] : B[N/x]$.

- *Stripping*

$$\begin{array}{rrcl}
(i) & \Gamma \vdash s : R,\ s \in \mathcal{S} & \Rightarrow & R =_\beta s' \textit{ with } s : s' \in \mathcal{A} \textit{ for some } s' \in \mathcal{S}, \\
(ii) & \Gamma \vdash x : R,\ x \in \mathsf{Var} & \Rightarrow & R =_\beta A \textit{ with } x : A \in \Gamma \textit{ for some term } A, \\
(iii) & \Gamma \vdash \Pi x{:}A.B : R & \Rightarrow & \Gamma \vdash A : s_1, \Gamma, x{:}A \vdash B : s_2 \textit{ and } R =_\beta s_3 \\
 & & & \textit{with } (s_1, s_2, s_3) \in \mathcal{R} \textit{ for some } s_1, s_2, s_3 \in \mathcal{S}, \\
(iv) & \Gamma \vdash \lambda x{:}A.M : R & \Rightarrow & \Gamma, x{:}A \vdash M : B, \Gamma \vdash \Pi x{:}A.B : s \textit{ and} \\
 & & & R =_\beta \Pi x{:}A.B \textit{ for some term } B \textit{ and } s \in \mathcal{S}, \\
(v) & \Gamma \vdash MN : R & \Rightarrow & \Gamma \vdash M : \Pi x{:}A.B, \Gamma \vdash N : A \textit{ with } R =_\beta B[N/x] \\
 & & & \textit{for some terms } A \textit{ and } B.
\end{array}$$

- *Subject Reduction*
  If $\Gamma \vdash M : A$ *and* $M \twoheadrightarrow_\beta N$*, then* $\Gamma \vdash N : A$.

- *Confluence*
  If $\Gamma \vdash M : A$, $\Gamma \vdash N : A$ *and* $M =_\beta N$*, then there is a term* Q *with* $\Gamma \vdash Q : A$ *and* $M \twoheadrightarrow_\beta Q$, $N \twoheadrightarrow_\beta Q$.

The definition of Pure Type System gives rise to an interesting notion of morphism between typed λ calculi which can be described by taking into account only the sorts, axioms and rules of the system.

Definition 2.5 *Let* $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ *and* $\lambda(\mathcal{S}',\mathcal{A}',\mathcal{R}')$ *be* PTS*s. A* morphism *from* $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ *to* $\lambda(\mathcal{S}',\mathcal{A}',\mathcal{R}')$ *is a mapping* f *from* $\mathcal{S}$ *to* $\mathcal{S}'$ *that* preserves axioms and rules*, that is*

$$\begin{array}{rcl}
s_1{:}s_2 \in \mathcal{S} & \Rightarrow & f(s_1){:}f(s_2) \in \mathcal{S}', \\
(s_1, s_2, s_3) \in \mathcal{R} & \Rightarrow & (f(s_1), f(s_2), f(s_3)) \in \mathcal{R}'.
\end{array}$$

A PTS-morphism f from $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ to $\lambda(\mathcal{S}',\mathcal{A}',\mathcal{R}')$ immediately extends to a mapping from the pseudoterms of $\lambda(\mathcal{S},\mathcal{A},\mathcal{R})$ to the pseudoterms of $\lambda(\mathcal{S}',\mathcal{A}',\mathcal{R}')$ and hence to a mapping from pseudocontexts to pseudocontexts. This mapping preserves substitution and β-equality and also derivability:

Lemma 2.6 *If* f *is a* PTS*-morphism from* ζ *to* ζ'*, then*

$$\Gamma \vdash_\zeta M : A \Rightarrow f(\Gamma) \vdash_{\zeta'} f(M) : f(A).$$

2.2 The cube of typed λ calculi

Definition 2.7 *The Barendregt's cube of typed* λ *calculi consists of eight* PTS*s. Each of them has*

$$\begin{array}{rcl}
\mathcal{S} & := & \{\star, \Box\}, \\
\mathcal{A} & := & \{\star : \Box\}.
\end{array}$$

The set of rules $\mathcal{R}$ for each system are as given in the following table.

$\lambda{\to}$	$(\star,\star)$			
$\lambda 2$	$(\star,\star)$	$(\Box,\star)$		
λP	$(\star,\star)$		$(\star,\Box)$	
$\lambda\overline{\omega}$	$(\star,\star)$			$(\Box,\Box)$
$\lambda\omega$	$(\star,\star)$	$(\Box,\star)$		$(\Box,\Box)$
$\lambda P2$	$(\star,\star)$	$(\Box,\star)$	$(\star,\Box)$	
$\lambda P\overline{\omega}$	$(\star,\star)$		$(\star,\Box)$	$(\Box,\Box)$
$\lambda P\omega$	$(\star,\star)$	$(\Box,\star)$	$(\star,\Box)$	$(\Box,\Box)$.

The system $\lambda P\omega$ is the Calculus of Constructions, sometimes called the Pure *Calculus of Constructions to distinguish it from its variants and extensions. We refer to it as CC. The systems of the cube are usually presented as follows.*

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega\ (= CC) \\
\lambda 2 \arrow[rr] \arrow[ur] & & \lambda P2 \arrow[ur] & \\
 & \lambda\overline{\omega} \arrow[rr] \arrow[uu] & & \lambda P\overline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[rr] \arrow[uu] \arrow[ur] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

where an arrow denotes inclusion of one system in another.

The systems $\lambda{\to}$ and $\lambda 2$ are also known as the simply typed lambda calculus and the polymorphically typed λ calculus (due to Girard, as system F, and Reynolds.) The system $\lambda\omega$ is a higher order version of $\lambda 2$, also known as Girard's system Fω. The presentation of these systems as a PTS is quite different from the original one. If one is just interested in those systems alone it is in general more convenient to study them in their original presentation. The PTS framework is more convenient for systems with *type dependency*, that is the feature that a type $A{:}\star$ may itself contain a term M with $M{:}B{:}\star$. This situation only occurs in the presence of the rule $(\star,\Box)$. In that case there is no other syntax for the systems which is essentially more convenient then the PTS format. The system λP is very close to the system LF, [Harper et al. 1987]. In fact LF is obtained from λP by replacing in the conversion rule the side condition $A =_\beta B$ by $A =_{\beta\eta} B$. The system λPω is the Calculus of Constructions, due to [Coquand 1985]. (See also [Coquand and Huet 1988].) The system λP2 was defined under the same name by [Longo and Moggi 1988].

The formulas-as-types embedding from logical systems into the systems of the cube is best understood by first defining a cube of eight 'logical typed λ calculi'. These are systems for which there is a clear one-to-one correspondence between the original logical system and the typed λ calculus. This correspondence is given by the formulas-as-types embedding. This embedding assigns to every formula φ a type $\tilde{\varphi}$ and to every proof in natural deduction style a term such that a proof of φ becomes a term of the type $\tilde{\varphi}$. That this embedding is one-to-one means that every term of the type $\tilde{\varphi}$ is the image of a proof of φ.

Definition 2.8 *The* logic cube *[Berardi 1990] consists of eight* PTS*s, each of them having as sorts and axioms*

$$\begin{array}{rcl} \mathcal{S} & = & \mathsf{Prop}, \mathsf{Set}, \mathsf{Type}^p, \mathsf{Type}^s, \\ \mathcal{A} & = & \mathsf{Prop} : \mathsf{Type}^p \mathsf{Set} : \mathsf{Type}^s. \end{array}$$

The rules of each of the systems is given by the following table

λPROP				
	$(\mathsf{Prop}, \mathsf{Prop})$			
λPROP2				
	$(\mathsf{Prop}, \mathsf{Prop})$		$(\mathsf{Type}^p, \mathsf{Prop})$	
$\lambda\mathrm{PROP}\overline{\omega}$				$(\mathsf{Type}^p, \mathsf{Type}^p)$,
	$(\mathsf{Prop}, \mathsf{Prop})$			
$\lambda\mathrm{PROP}\omega$				$(\mathsf{Type}^p, \mathsf{Type}^p)$
	$(\mathsf{Prop}, \mathsf{Prop})$		$(\mathsf{Type}^p, \mathsf{Prop})$	
λPRED	$(\mathsf{Set}, \mathsf{Set})$	$(\mathsf{Set}, \mathsf{Type}^p)$		
	$(\mathsf{Prop}, \mathsf{Prop})$	$(\mathsf{Set}, \mathsf{Prop})$		
λPRED2	$(\mathsf{Set}, \mathsf{Set})$	$(\mathsf{Set}, \mathsf{Type}^p)$		
	$(\mathsf{Prop}, \mathsf{Prop})$	$(\mathsf{Set}, \mathsf{Prop})$	$(\mathsf{Type}^p, \mathsf{Prop})$	
$\lambda\mathrm{PRED}\overline{\omega}$	$(\mathsf{Set}, \mathsf{Set})$	$(\mathsf{Set}, \mathsf{Type}^p)$	$(\mathsf{Type}^p, \mathsf{Set})$	$(\mathsf{Type}^p, \mathsf{Type}^p)$
	$(\mathsf{Prop}, \mathsf{Prop})$	$(\mathsf{Set}, \mathsf{Prop})$		
$\lambda\mathrm{PRED}\omega$	$(\mathsf{Set}, \mathsf{Set})$	$(\mathsf{Set}, \mathsf{Type}^p)$	$(\mathsf{Type}^p, \mathsf{Set})$	$(\mathsf{Type}^p, \mathsf{Type}^p)$
	$(\mathsf{Prop}, \mathsf{Prop})$	$(\mathsf{Set}, \mathsf{Prop})$	$(\mathsf{Type}^p, \mathsf{Prop})$	

The systems are presented in a picture as follows.

$$
\begin{tikzcd}
 & \lambda\mathrm{PROP}\omega \arrow[rr] & & \lambda\mathrm{PRED}\omega \\
\lambda\mathrm{PROP}2 \arrow[rr] \arrow[ur] & & \lambda\mathrm{PRED}2 \arrow[ur] & \\
 & \lambda\mathrm{PROP}\overline{\omega} \arrow[rr] \arrow[uu] & & \lambda\mathrm{PRED}\overline{\omega} \arrow[uu] \\
\lambda\mathrm{PROP} \arrow[rr] \arrow[uu] \arrow[ur] & & \lambda\mathrm{PRED} \arrow[uu] \arrow[ur] &
\end{tikzcd}
$$

where an arrow denotes inclusion of one system in another.

That the type systems described above indeed correspond to logical systems will not be discussed here. For details we refer to [Barendregt 1992], [Tonino anf Fujita 1992] and [Geuvers 1993]. To get the idea we give some examples.

Examples 2.9 *1.* $A{:}\mathsf{Set}, R{:}A{\to}A{\to}\mathsf{Prop}, \varphi{:}\mathsf{Prop} \vdash$ $\lambda p{:}(\Pi x,y{:}A.Rxy{\to}Ryx{\to}\varphi).\lambda x{:}A.\lambda q{:}Rxx.pxxqq : (\Pi x,y{:}A.Rxy{\to}Ryx{\to}\varphi){\to}(\Pi x{:}A.Rxx{\to}\varphi)$ *in* λPRED. *The term* $R : A{\to}A{\to}\mathsf{Prop}$ *is understood as a binary relation on A. It should be clear how the term* $\lambda x{:}A.\lambda q{:}Rxx.pxxqq$ *corresponds to a proof in natural deduction style of the proposition* $(\forall x,y \in A[R(x,y) \supset R(y,x) \supset \varphi]) \supset (\forall x \in A[R(x,x){\to}\varphi])$.

2. *In any system that contains* λPROP2, $\perp$ *can be defined as* $\Pi\alpha{:}\mathsf{Prop}.\alpha(:\mathsf{Prop})$. *One has indeed* $\varphi{:}\mathsf{Prop} \vdash \lambda p{:}\perp.p\varphi : \perp{\to}\varphi$.

3. *In* λPRED2 *one has* $A{:}\mathsf{Set} \vdash$ $\lambda R{:}A{\to}A{\to}\mathsf{Prop}.\lambda p{:}(\Pi x,y{:}A.Rxy{\to}Ryx{\to}\perp).\lambda x{:}A.\lambda q{:}Rxx.pxxqq :$ $\Pi R{:}A{\to}A{\to}\mathsf{Prop}.(\Pi x,y{:}A.Rxy{\to}Ryx{\to}\perp){\to}(\Pi x{:}A.Rxx{\to}\perp)$, *stating that any binary relation that is antisymmetric is areflexive.*

The systems of the Barendregt's cube and the logic cube enjoy some more special properties that will be used. First of all, the type of a term is unique up to β-conversion. (The proof is by induction on terms, see [Geuvers 1993] or [Barendregt 1992].)

Proposition 2.10 (Uniqueness of Types) *For a system in one of the two cubes one has that if* $\Gamma \vdash M : A$ *and* $\Gamma \vdash M : B$, *then* $A =_\beta B$.

Another nice thing is that, if variables are treated with some care, then the terms can be classified into disjoint sets. One therefore devides the set of variables Var into disjoint subsets Var^s ($s \in \mathcal{S}$). In the rules (weak) and (var), if $\Gamma \vdash A : s$ is the premise, one can now only take a variable x from the set Var^s. In the type systems of the Barendregt's cube, one often uses greek characters and capitals for the variables in $\mathsf{Var}^\Box$ and latin characters for the variables in $\mathsf{Var}^\star$. The following definition is now useful.

Definition 2.11 *We work in some system of the Barendregt's cube.*

1. *The set of kinds is defined by* $\mathsf{Kind} := \{A \mid \exists\Gamma[\Gamma \vdash A : \Box]\}]$.
2. *The set of types is defined by* $\mathsf{Type} := \{A \mid \exists\Gamma[\Gamma \vdash A : \star]\}$.
3. *The set of constructors is defined by* $\mathsf{Constr} := \{P \mid \exists A, \Gamma[\Gamma \vdash P : A : \Box]\}$.
4. *The set of objects is defined by* $\mathsf{Obj} := \{P \mid \exists A, \Gamma[\Gamma \vdash P : A : \star]\}$.

Here $\Gamma \vdash P : A : \star$ *denotes the fact that* $\Gamma \vdash P : A$ *and* $\Gamma \vdash A : \star$.

The usefulness of this definition is due to the following lemma. (For a detailed proof see [Geuvers 1993].)

Lemma 2.12 (Classification) *We work in some system of the Barendregt's cube.*

$$\begin{aligned} \mathsf{Kind} \cap \mathsf{Type} &= \emptyset, \\ \mathsf{Constr} \cap \mathsf{Obj} &= \emptyset. \end{aligned}$$

The formulas-as-types embedding of a logic into the corresponding system of the logic cube is an isomorphism (for details see [Geuvers 1993]), so we can restrict our study of the formulas-as-types embedding into the systems of the Barendregt's cube to the study of the *collapsing mapping* H that maps the systems of the logic cube into the ones of the cube of typed λ calculi.

Definition 2.13 *The collapsing mapping* H *is defined as the family of* PTS*-morphisms from logic cube to Barendregt's cube given by*

$$\begin{aligned} H(\mathsf{Prop}) &= \star, \\ H(\mathsf{Set}) &= \star, \\ H(\mathsf{Type}^p) &= \Box, \\ H(\mathsf{Type}^s) &= \Box. \end{aligned}$$

It is immediate that the collapsing mapping H does not really do anything for the systems of the left plane of the cube. The sorts Prop and Type^p are renamed as $\star$ and $\Box$, but there are no additional rules. This implies that the formulas-as-types embedding from propositional logics into a system of the left plane of the cube is an isomorphism. For the right plane of the cube the situation is more interesting, because the sorts Prop and Set, respectively Type^p and Type^s are mapped to the same sort in the Barendregt's cube. This question of *completeness* of H becomes a real issue here.

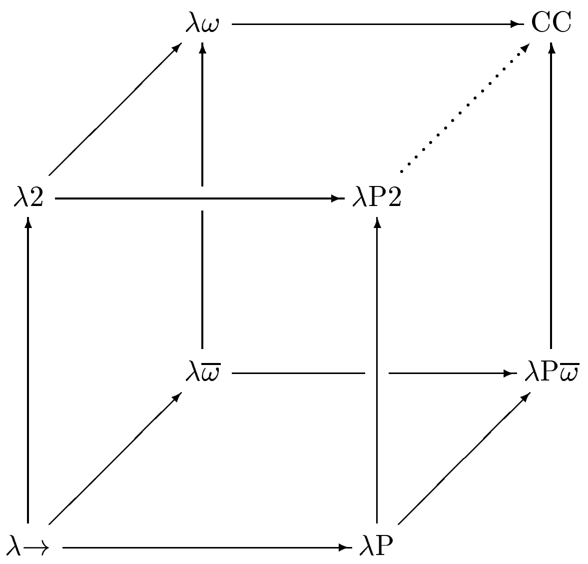
Definition 2.14 *For* L_i *a system of the logic cube and* S_i *the corresponding system in the Barendregt's cube, we say that* $H : L_i \to S_i$ *is* complete *if for all context* Γ *in* L_i *and* φ *with* $\Gamma \vdash_{L_i} \varphi : \mathsf{Prop}$, *one has*

$$H(\Gamma) \vdash_{S_i} M : H(\varphi) \Rightarrow \exists N[\Gamma \vdash_{L_i} N : \varphi].$$

Completeness is of course important because typed λ calculi like the Calculus of Constructions are intended to be used as systems for formalizing mathematics, which is done by reasoning in the embedded higher order order predicate logic. However, it is well-known by now that the embedding of higher order predicate logic into CC is not complete. In contrast, the embedding H from λPRED into λP is complete and for the embedding H of λPRED2 into λP2 the question of completeness is still open. (The incompleteness of $H : \lambda\text{PRED}\omega \to \text{CC}$ was first noticed by [Berardi 1989] and [Geuvers 1989]. See also [Barendregt 1992] and [Geuvers 1993]. The completeness of $H : \lambda\text{PRED} \to \lambda\text{P}$ is proved in [Berardi 1989] and [Barendsen and Geuvers 1989]. See also [Geuvers 1993].)

3 Conservativity relations inside the cube

We now want to address the question of conservativity inside the cube of typed λ calculi and the logic cube. We first look at the cube of typed λ calculi, because the situation for the logic cube is very similar. There are four results that do the whole job, resulting in the following picture.



where an arrow denotes a conservative inclusion and a dotted arrow denotes a non-conservative inclusion. By transitivity of conservativity (if system 3 is conservative over system 2 and system 2 is conservative over system 1, then system 3 is conservative over system 1), it is no problem to fill in the picture further. (Draw the arrows between two non-adjacent systems) We can collect all this in the following Proposition.

Theorem 3.1 *For S_1 and S_2 two systems in the cube of typed lambda calculi such that $S_1 \subseteq S_2$:*

$$S_2 \textit{ is conservative over } S_1 \Leftrightarrow S_2 \neq CC \ \& \ S_1 \neq \lambda P2.$$

Proof It suffices to prove the following four results.

1. If $S_2 \supseteq S_1$, S_1 a system of the lower plane in the cube, then S_2 is conservative over S_1.(Proposition 3.2.)

2. If S_2 a system in the right plane of the cube, S_1 the adjacent system in the left plane, then S_2 is conservative over S_1.(Proposition 3.6.)

3. $\lambda P\omega$ is not conservative over $\lambda P2$,

4. $\lambda\omega$ is conservative over $\lambda 2$. (Corollary 4.21.)

The fourth is a consequence of Corollary 4.21, saying that PROPω is conservative over PROP2 and of the fact that PROPω and PROP2 are isomorphic to, respectively, $\lambda\omega$ and $\lambda 2$ via the formulas-as-types embedding. The conservativity of PROPω over PROP2 will be proved in detail later by using semantical methods.
The third was verfied in detail by [Ruys 1991], following an idea from Berardi. The idea is to look at a context Γ in $\lambda P2$ that represents Arithmetic. Then Γ with $\lambda P2$ is as strong as second order Arithmetic and Γ with $\lambda P\omega$ is as strong as higher order Arithmetic. Hence we can use Gödel's Second Incompleteness Theorem to show that in $\lambda P2$ one can not derive from Γ that Γ is consistent in $\lambda P2$. On the other hand in $\lambda P\omega$ one can derive from Γ that Γ is consistent in $\lambda P2$. Hence the non-conservativity. $\Box$

We first prove the Proposition about conservativity of systems over systems in the lower plane. The Proposition was also proved in [Verschuren 1990] in a slightly different way.

Proposition 3.2 *Let S_1 be a system of the lower plane and S_2 be any system of the cube such that $S_1 \subseteq S_2$. Then*

$$\left.\begin{array}{r}\Gamma \vdash_{S_1} B : \star \\ \Gamma \vdash_{S_2} M : B \\ \Gamma \text{ and } M \text{ in normal form}\end{array}\right\} \Rightarrow \Gamma \vdash_{S_1} M : B.$$

Proof By induction on the structure of M.

applic. Say $M \equiv xP_1 \cdots P_n$. Then, by Stripping, $x{:}A \in \Gamma$ with $A =_\beta \Pi y_1{:}C.D$ for some C and D. Now, A is in normal form (because Γ is) and so A is itself a Π-term, say $A \equiv \Pi y_1{:}C_1.D_1$. So, $x{:}\Pi y_1{:}C_1.D_1 \in \Gamma$. Now, $\Gamma \vdash_{S_1} \Pi y_1{:}C_1.D_1 : \star$ (in the lower plane, i.e. without the rule $(\Box,\star)$), but then also

$$\Gamma \vdash_{S_1} C_1 : \star.$$

Of course we also have

$$\Gamma \vdash_{S_2} P_1 : C_1,$$

so by IH (note that P_1 is in normal form), $\Gamma \vdash_{S_1} P_1 : C_1$. Hence $\Gamma \vdash_{S_1} xP_1 : D_1[P_1/y_1]$. We can now go further with P_2: We know that $D_1[P_1/y_1] \twoheadrightarrow_\beta \Pi y_2{:}C_2.D_2$. Now, $\Gamma \vdash_{S_1} D_1[P_1/y_1] : \star$ and hence $\Gamma \vdash_{S_1} \Pi y_2{:}C_2.D_2 : \star$ by Subject Reduction. So

$$\Gamma \vdash_{S_1} C_2 : \star.$$

Also

$$\Gamma \vdash_{S_2} P_2 : C_2,$$

so again we can apply IH to obtain $\Gamma \vdash_{S_1} P_2 : C_2$ and hence $\Gamma \vdash_{S_1} xP_1P_2 : D_2[P_2/y_2]$. Continuing in this way upto n we find that $\Gamma \vdash_{S_1} xP_1 \cdots P_n : D_n[P_n/y_n]$ with $D_n[P_n/y_n] =_\beta B$. By one application of the conversion rule (using $\Gamma \vdash_{S_1} B : \star$) we conclude $\Gamma \vdash_{S_1} xP_1 \cdots P_n : B$.

abstr. Say $M \equiv \lambda x{:}A.N$. Then $B \twoheadrightarrow_\beta \Pi x{:}A.C$ for some C (note that A is in normal form). So $\Gamma \vdash_{S_1} \Pi x{:}A.C : \star$ by Subject Reduction and $\Gamma, x{:}A \vdash_{S_2} M : C$ (by Stripping and the conversion rule). By IH we conclude $\Gamma, x{:}A \vdash_{S_1} M : C$. Now we are done: By one λ-abstraction and one conversion we conclude $\Gamma \vdash_{S_1} \lambda x{:}AM : B$. $\Box$

The side condition Γ in normal form has just been added for convenience (in giving the proof.) It is not essential and it may be dropped.

As a corollary one finds that a system of the cube is conservative over all its subsystems of the lower plane. If $S_1 \subset S_2$, S_1 in the lower plane, then

$$\left.\begin{array}{r}\Gamma \vdash_{S_1} A : \mathsf{Prop}\\ \Gamma \vdash_{S_2} M : A\end{array}\right\} \Rightarrow \exists N[\Gamma \vdash_{S_1} N : A].$$

This can even be made more precise, because the term N can be computed directly from the term M as follows.

Corollary 3.3

$$\left.\begin{array}{r}\Gamma \vdash_{S_1} A : \mathsf{Prop}\\ \Gamma \vdash_{S_2} M : A\end{array}\right\} \Rightarrow \Gamma \vdash_{S_1} \mathit{nf}(M) : A,$$

where $\mathit{nf}(M)$ denotes the β-normal form of M.

We now prove the conservativity of the right plane over the left plane. The idea is to define a mapping that removes all type dependencies. This mapping will go from a system in the right plane to the adjacent system in the left plane and is the identity on terms that are already well-typed in the left plane. Hence the conservativity. The proof is originally independently due to [Paulin 1989] and [Berardi 1990]. The first described the mapping from $\lambda\mathrm{P}\omega$ to $\lambda\omega$ in the first place to use it for program extraction; the second described the collection of four mappings (which is a straightforward generalisation of the mapping from $\lambda\mathrm{P}\omega$ to $\lambda\omega$) to give a conservativity proof. The mappings are very much related to similar mappings one can define from predicate logic to propositional logic to prove conservativity of the first over the second.

Definition 3.4 ([Paulin 1989] and [Berardi 1990]) *Let S_2 be a system of the right plane and S_1 the adjacent system in the left plane. The mapping $[-] : \mathsf{Term}(S_2) \to \mathsf{Term}(S_1)$ is defined as follows.*

$$\begin{array}{rcl}
[\Box] &=& \Box,\\
[\star] &=& \star,\\
[x] &=& x,\ \textit{for } x \textit{ a variable},\\
[\Pi x{:}A.B] &=& [B]\ \textit{if } A{:}\star, B{:}\Box,\\
&=& \Pi x{:}[A].[B]\ \textit{else},\\
[\lambda x{:}A.M] &=& [M]\ \textit{if } A{:}\star, M{:}B{:}\Box,\ \textit{(for some } B\textit{)},\\
&=& \lambda x{:}[A].[M]\ \textit{else},\\
[PM] &=& [P]\ \textit{if } M{:}A{:}\star, P{:}B{:}\Box,\ \textit{(for some } A, B\textit{)},\\
&=& [P][M]\ \textit{else},
\end{array}$$

Remark 3.5 *The side conditions in the defintion are justified by the Classification Lemma (2.12).*

The mapping $[-]$ extends straightforwardly to contexts. The following proposition justifies the statement in the definition that the mapping $[-]$ goes from the right plane to the left plane.

Proposition 3.6 ([Paulin 1989] and [Berardi 1990]) *Let S_2 be a system in the right plane and S_1 the adjacent system in the left plane of the cube.*

$$\Gamma \vdash_{S_2} M : A \Rightarrow [\Gamma] \vdash_{S_1} [M] : [A]$$

Proof By a straightforward induction on the derivation of $\Gamma \vdash_{S_2} M : A$. $\Box$

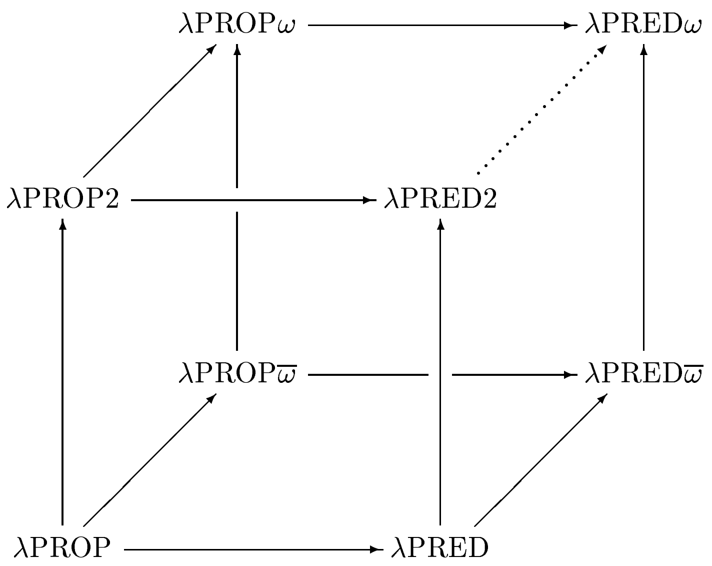
Corollary 3.7 ([Paulin 1989],[Berardi 1990]) *For S_2 a system in the right plane and S_1 the adjacent system in the left plane of the cube we have that S_2 is conservative over S_1.*

Proof The only thing to check is that for $M \in \mathsf{Term}(S_1)$, $[M] \equiv M$. This is done by an easy induction on the structure of M. $\Box$

Corollary 3.7 can be made a bit more precise by stating how the term in the smaller system is computed from the term in the larger system. For S_2 in the right plane and S_1 the adjacent system in the left plane one has

$$\left.\begin{array}{r}\Gamma \vdash_{S_1} A : \mathsf{Prop} \\ \Gamma \vdash_{S_2} M : A\end{array}\right\} \Rightarrow \Gamma \vdash_{S_1} [M] : A,$$

The conservativity relations in the logic cube (Definition 2.8) are as follows. (An arrow denotes a conservative extension, a dotted arrow a non-conservative extension.)



Proposition 3.8 *For L_1 and L_2 two systems in the logic cube such that $L_1 \subseteq L_2$:*

$$L_2 \textit{ is conservative over } L_1 \Leftrightarrow L_2 \neq \lambda\text{PRED}\omega \;\&\; L_1 \neq \lambda\text{PRED2}.$$

Proof The proof is completely analoguous to the proof for the cube of typed lambda calculi. In fact, what has to be proved for conservativity is that, if $L_1 \subset L_2$ and Γ is a context of L_1 with $\Gamma \vdash A : \mathsf{Prop}$, then

$$\Gamma \vdash_{L_2} M : A \Rightarrow \exists N[\Gamma \vdash_{L_1} N : A].$$

The proof of non-conservativity of λPREDω over λPRED2 is the same as for CC over λP2, by using Gödel's incompleteness theorem. The proof of conservativity of any system over a subsystem in the lower plane is again by normalization (of the proof terms).
The proof of conservativity of the right plane over the left plane can be done by defining a mapping that forgets predicates, analogously to the one defined in Definition 3.4. A slightly easier proof, for which we don't have to define any mapping at all is the following: Let Γ be a context of L_1 in the left plane and let A be a term such that $\Gamma \vdash_{L_1} A : \mathsf{Prop}$. Let L_2 be the adjacent system in the right plane and let $\Gamma \vdash_{L_2} M : A$. If S_1 is the system in the Barendregt's cube that corresponds with L_1 and S_2 is the system in the Barendregt's cube that corresponds with L_2, then $H(\Gamma) \vdash_{S_2} H(M) : H(A)$ and hence $\exists N[H(\Gamma) \vdash_{S_1} N : H(A)]$ by the conservativity in the Barendregt's cube. Because the formulas-as-types embedding H is an isomorphism on the left plane of the cube, we can conclude that $\exists N[\Gamma \vdash_{L_1} N : A]$.
The proof of conservativity of λPROPω over λPROP2 is given in the following section. □

4 The conservativity of $\lambda\omega$ over $\lambda 2$

The conservativity of $\lambda\omega$ over $\lambda 2$ is shown by proving that PROPω is conservative over PROP2. The conservativity of $\lambda\omega$ over $\lambda 2$ then follows from the fact that the formulas-as-types embedding is an isomorphism. In order to be very specific about the conservativity proof, we first give the detailed syntax of the systems of second and higher order propositional logic.

4.1 second and higher order propositional logics

Definition 4.1 *For n a natural number, the system of nth order propositional logic, notation* PROPn *is defined by first giving the nth order language and then describing the deduction rules for the nth order system as follows.*

1. *The* domains *are given by*
$$\mathcal{D} ::= \mathsf{Prop} \,|\, (\mathcal{D} \to \mathcal{D}).$$
We let the brackets associate to the right, so $\mathsf{Prop}\to(\mathsf{Prop}\to\mathsf{Prop})$ *will be denoted by* $\mathsf{Prop}\to\mathsf{Prop}\to\mathsf{Prop}$ *and so every domain can be written as* $D_1 \to \ldots \to D_p \to \mathsf{Prop}$, *with* $D_1, \ldots, D_p$ *domains.*

2. *The* order of a domain D, $\mathsf{ord}(D)$, *is defined by*
$$\begin{aligned} \mathsf{ord}(\mathsf{Prop}) &= 2, \\ \mathsf{ord}(D_1 \to \ldots \to D_p \to \mathsf{Prop}) &= max\{\mathsf{ord}(D_i) \,|\, 1 \leq i \leq p\} + 1. \end{aligned}$$
The orders are defined in such a way that in n-th order logic one can quantify over domains of order $\leq n$. Hence Prop *is of order 2, because in second order propositional logic one can quantify over the set of all formulas. The domain* $\mathsf{Prop}\to\mathsf{Prop}$ *should be understood as the collection of sets of formulas (truth values), identifying a function P from* Prop *to* Prop *with the set of formulas φ for which $P\varphi$ holds.*

3. *For n a fixed positive natural number, the terms of the nth order language are defined as follows. (Each term is an element of a specific domain, which relation is denoted by ϵ).*
    - *There are countably many variables of domain D for any D with* $\mathsf{ord}(D) \leq n$,

- *If* $M \,\epsilon\, D_2$*,* x *a variable of domain* D_1 *and* $\mathsf{ord}(D_1{\to}D_2) \leq n$*, then* $\lambda x \epsilon D_1.M \,\epsilon\, D_1{\to}D_2$*,*
- *If* $M \,\epsilon\, D_1{\to}D_2$*,* $N \,\epsilon\, D_1$*, then* $MN \,\epsilon\, D_2$*,*
- *If* $\varphi \,\epsilon\, \mathsf{Prop}$*,* x *a variable of domain* D *with* $\mathsf{ord}(D) \leq n$*, then* $\forall x \epsilon D.\varphi \,\epsilon\, \mathsf{Prop}$.
- *If* $\varphi \,\epsilon\, \mathsf{Prop}$ *and* $\psi \,\epsilon\, \mathsf{Prop}$*, then* $\varphi \supset \psi \,\epsilon\, \mathsf{Prop}$*.*

4. *The terms* φ *for which* $\varphi \,\epsilon\, \mathsf{Prop}$ *are called* formulas *and* Form *denotes the set of formulas.*

5. *On the terms we have the well-known notion of definitional equality by* β*-conversion. This equality is denoted by* $=$*. The definitional equality allows us to identify for example the application of the function* $\lambda x{:}\mathsf{Prop}.x \supset x$ *(of domain* $\mathsf{Prop}{\to}\mathsf{Prop}$*) to* φ *with the a formula* $\varphi \supset \varphi$*.*

6. *For n a specific positive natural number, we now describe the deduction rules of the nth order predicate logic (in natural deduction style) that allow us to build derivations. So in the following let* φ *and* ψ *be formulas of the nth order language.*

$$(\supset\text{-}I)\quad \frac{\begin{array}{c}[\varphi]^i\\ \vdots\\ \psi\end{array}}{\varphi \supset \psi}\,i \qquad\qquad (\supset\text{-}E)\quad \frac{\varphi \supset \psi \quad \varphi}{\psi}$$

$$(\forall\text{-}I)\quad \frac{\psi}{\forall x \epsilon D.\psi}(*) \qquad\qquad (\forall\text{-}E)\quad \frac{\forall x \epsilon D.\psi}{\psi[t/x]}\ \textit{if } t \,\epsilon\, D$$

$$(conv)\quad \frac{\psi}{\varphi}\ \textit{if } \varphi = \psi$$

The formula occurrences that are between brackets ([−]) in the $\supset$*-I rule are* discharged*. The superscript i in the* $\supset$*-I rule is taken from a countable set of indices* I*. The index i uniquely corresponds to one specific application of the* $\supset$*-I rule, so we do not allow one index to be used more than once. The use of the indexes allows us to fix those formula occurrences that are discharged at a specific application of the* $\supset$*-I rule.*
$(*)$*: in the* $\forall$*-I rule we make the usual restriction that the variable* x *may not occur free in a non-discharged assumption of the derivation.*
For Γ *a set of formulas of* PROPn *and* φ *a formula of* PROPn*, we say that* φ is derivable from Γ in PROPn*, notation* $\Gamma \vdash_{\mathrm{PROP}n} \varphi$*, if there is a derivation with root* φ *and all non-discharged formulas in* Γ*.*

The system of higher order proposition logic, notation PROPω*, is the union of all* PROPn*.*

Remark 4.2 *The choice for the connectives* $\supset$ *and* $\forall$ *may seem minimal. It is however a well-known fact that in second and higher order systems, the intuitionistic connectives* &, $\vee$, $\neg$ *and* $\exists$ *can be defined in terms of* $\supset$ *and* $\forall$ *as follows. (Let* φ *and* ψ *be formulas).*

$$\begin{aligned}\varphi \,\&\, \psi &:= \forall \alpha \epsilon \mathsf{Prop}.(\varphi \supset \psi \supset \alpha) \supset \alpha,\\ \varphi \vee \psi &:= \forall \alpha \epsilon \mathsf{Prop}.(\varphi \supset \alpha) \supset (\psi \supset \alpha) \supset \alpha,\end{aligned}$$

$$\begin{array}{rcl}\bot & := & \forall\alpha\epsilon\mathsf{Prop}.\alpha,\\ \neg\varphi & := & \varphi\supset\bot,\\ \exists x\,\epsilon\,D\varphi & := & \forall\alpha\epsilon\mathsf{Prop}.(\forall x\epsilon D.\varphi\supset\alpha)\supset\alpha.\end{array}$$

Similarly we can define an equality judgement (the β-equality $=$, the definitional equality of the language*, is purely syntactical) by taking the so called Leibniz equality: for $t, q\,\epsilon\,D$,*

$$t =_D q := \forall P\epsilon D{\rightarrow}\mathsf{Prop}.Pt \supset Pq,$$

which says that two objects are equal if they satisfy the same properties. (It is not difficult to show that $=_D$ is symmetric).

It is not difficult to check that all the standard logical rules hold for $\&, \vee, \bot, \neg, \exists$ and $=$. In the following we shall freely use these symbols.

Remark 4.3 *In each* PROPn *($n \geq 2$), the* comprehension *property is satisfied. That is, for all $\varphi(\vec{x})$: Prop with $\vec{x} = x_1, \ldots, x_p$ a sequence of free variables, possibly occurring in φ ($x_i\,\epsilon\,D_i$), we have*

$$\exists P\,\epsilon\,D_1{\rightarrow}\cdots D_p{\rightarrow}\mathsf{Prop}.\forall\vec{x}\epsilon\vec{D}(\varphi \leftrightarrow Px_1\cdots x_p).$$

(Take $P \equiv \lambda x_1\,\epsilon\,D_1.\ldots\lambda x_p\,\epsilon\,D_p.\varphi(\vec{x})$.)

The presentation of propositional logics above can be extended to predicate logics, which is done in [Geuvers 1993]. There also the classical variants of the systems are studied. Here we restrict to the constructive versions of the propositional logics, because PROPω and PROP2 are the ones that correspond to the type systems $\lambda\omega$ and F.

4.2 Extensionality

The definitional equality on the terms is β-equality. There is no objection to taking $\beta\eta$-equality instead: all the properties remain to hold. In fact it would make a lot of sense to do so, because we tend to view λ-abstraction as the necessary mechanism to make comprehension work. (And so both $P\,\epsilon\,\mathsf{Prop}{\rightarrow}\mathsf{Prop}$ and $\lambda x\,\epsilon\,\mathsf{Prop}.Px$ describe the collection of formulas φ for which $P\varphi$ holds).

This is related to the issue of *extensionality*: terms of domain $D{\rightarrow}\mathsf{Prop}$ are to be understood as predicates on D or also as subsets of D, an element t being in the set $P\,\epsilon\,D{\rightarrow}\mathsf{Prop}$ if Pt holds. But if we take this set-theoretic understanding seriously, we have to identify predicates that are extensionally equal:

$$(\forall\vec{x}.f\vec{x} \supset g\vec{x}\ \&\ g\vec{x} \supset f\vec{x}) \supset f =_D g. \quad (1)$$

Of course, this formula is in general not provable in our systems. However, in the standard models where predicates are interpreted as real sets, the formula is satisfied, so it is an important extension. A difficulty is, that extensionality in the form of (1) is in general not even expressible: in PROPn we can not express extensionality for f and g of domain D if $\mathsf{ord}(D) = n$, because $f =_D g$ is not a formula of PROPn (it uses a quantification over $D{\rightarrow}\mathsf{Prop}$). This means that we shall have to express extensionality by a schematic rule.

Definition 4.4 *The* extensionality scheme*, (EXT), is*

$$(EXT)\frac{f\vec{x} \supset g\vec{x} \quad g\vec{x} \supset f\vec{x} \quad \varphi(f)}{\varphi(g)}\ (*)$$

where f and g are arbitrary terms of the same domain $D_1 \to \cdots \to D_n \to \mathsf{Prop}$ and $\varphi(f)$ stands for a formula φ with a specific marked occurrence of f. $(*)$ signifies the usual restriction that the variables of $\vec{x}$ may not occur free in a non-discharged assumption of the derivations of $f\vec{x} \supset g\vec{x}$ and of $g\vec{x} \supset f\vec{x}$.
The extension of a system with the rule (EXT) will be denoted by adding the prefix E-, so E-PROPn *is extensional nth order propositional logic.*

Notation 4.5 *For $f, g \,\epsilon\, D = D_1 \to \cdots \to D_n \to \mathsf{Prop}$, if quantification over $D_1, \ldots, D_n$ is allowed in the system we can compress the first two premises in the rule (EXT) to $\forall \vec{x}. f\vec{x} \supset g\vec{x}$ & $g\vec{x} \supset f\vec{x}$. For convenience this will also be denoted by $f \sim_D g$, so*

$$f \sim_D g := \forall \vec{x}. f\vec{x} \supset g\vec{x} \,\&\, g\vec{x} \supset f\vec{x},$$

where the D will usually be omitted if it is clear from the context.

Lemma 4.6 *The extensionality scheme for $D = \mathsf{Prop}$ is admissible in any of the propositional logics, i.e.*

$$\varphi \supset \psi, \psi \supset \varphi, \chi(\varphi) \vdash \chi(\psi)$$

is always provable.

Proof By an easy induction on the structure of χ. □

The following is now immediate by the fact that in PROP2 the only extensionality scheme that can be expressed is the one for $D = \mathsf{Prop}$.

Corollary 4.7 *In the system* E-PROP2 *of extensional second order propositional logic one can prove the same as in* PROP2*. That is*

$$\Gamma \vdash_{\text{E-PROP2}} \varphi \Leftrightarrow \Gamma \vdash_{\text{PROP2}} \varphi.$$

4.3 Algebraic semantics for intuitionistic propositional logics

In this section we describe a semantics for our systems of intuitionistic propositional logic in terms of Heyting algebras. It is well-known how this is done for the full first order propositional logic, giving rise to a completeness result. For second and higher order propositional logic we need to refine the notion of Heyting algebra to also allow interpretations for the universal quantifier. It is easily seen that *complete* Heyting algebras are strong enough to satisfy our purpose: complete Heyting algebras have arbitrary meets and joins, so for example $\forall f \,\epsilon\, \mathsf{Prop} \to \mathsf{Prop}.\varphi$ can be interpreted as $\bigwedge\{[\![\varphi]\!]_{[f:=F]} \,|\, F \in A \to A\}$. It is however not so easy to show the completeness of complete Heyting algebras over E-PROPn (for any n), because the Lindenbaum algebra defined from E-PROPn is not a complete Heyting algebra. The way out was suggested by Theorem 13.6.13 of [Troelstra and Van Dalen 1988], stating that any Heyting algebra can be embedded in a complete Heyting algebra such that $\supset$, $\perp$ and all existing $\bigvee$ and $\bigwedge$ are preserved (and hence the ordering is preserved). The embedding i that is constructed in the proof is also faithful with respect to the ordering, that is, if $i(a) \leq i(b)$ in the image, then $a \leq b$ in the original Heyting algebra. All this implies completeness of complete Heyting algebras with respect to E-PROPn, for any n. Hence we have conservativity of E-PROP$(n+1)$ over E-PROPn.

At this point we do not know how (if at all possible) to conclude the conservativity of PROP$(n+1)$ over PROPn from the conservativity of E-PROP$(n+1)$ over E-PROPn. However, we do have the conservativity of PROPn over PROP2 for any n, because PROP2 and E-PROP2 are the same system (Corollary 4.7).

It is obvious that extensionality is required in the syntax because the model notion is extensional: if, for example, $F, G : A \to A$ (where A is the carrier set of the algebra) and $F(a) = G(a)$ for all $a \in A$, then $F = G$.

The method of showing conservativity by semantical means seems to be quite essential here. Syntactic conservativity proofs (like the other in this paper) use mappings from the 'larger' system to the 'smaller' system that are the identity on the smaller system. These mappings also constitute a mapping from derivations to derivations that is the identity on derivations of the smaller system. This was the case for the proof of conservativity of the upper plane of the cube over the lower plane, where the proof-term in the smaller system is just obtained by normalizing the proof-term in the larger system. For the case of propositional logics, this method is impossible: there are formulas of PROP2 that have more and more cut-free derivations when we go higher in the hierarchy of propositional logics.

Definition 4.8 *A* Heyting algebra *(or just Ha) is a tuple* $(A, \wedge, \vee, \perp, \supset)$ *such that* $(A, \wedge, \vee)$ *is a lattice with least element* $\perp$ *and* $\supset$ *is a binary operation with*

$$a \wedge b \leq c \Leftrightarrow a \leq b \supset c.$$

Remember that $(A, \wedge, \vee)$ is a lattice if the binary operations $\wedge$ and $\vee$ satisfy the following requirements.

$$\begin{array}{rclrcl}
a \wedge a & = & a, & a \vee a & = & a, \\
a \wedge b & = & b \wedge a, & a \vee b & = & b \vee a, \\
a \wedge (b \wedge c) & = & (a \wedge b) \wedge c, & a \vee (b \vee c) & = & (a \vee b) \vee c, \\
a \vee (a \wedge b) & = & a, & a \wedge (a \vee b) & = & a.
\end{array}$$

Another way of defining the notion of lattice is by saying that it is a poset $(A, \leq)$ with the property that each pair of elements $a, b \in A$ has a least upperbound (denoted by $a \vee b$) and a greatest lowerbound (denoted by $a \wedge b$). By defining $a \leq b := a \wedge b = a$ we can then show the equivalence of the two definitions of lattice.

Definition 4.9 *A* complete Heyting algebra *(cHa) is a tuple* $(A, \bigwedge, \bigvee, \perp, \supset)$ *such that* $(A, \bigwedge, \bigvee)$ *is a complete lattice and* $(A, \wedge, \vee, \perp, \supset)$ *is a Heyting algebra. (So* $\bigvee$ *and* $\bigwedge$ *are mappings from* $\wp(A)$ *to* A *such that for* $X \subset A$, $\bigvee X$ *is the least upperbound of* X *and* $\bigwedge X$ *is the greatest lower bound of* X. *The binary operations* $\wedge$ *and* $\vee$ *are defined by (for* $a, b \in A$*)* $a \wedge b := \bigwedge\{a, b\}$ *and* $a \vee b := \bigvee\{a, b\}$*).*

An important feature of Heyting algebras which is forced upon by the presence of the binary operation $\supset$, is that they satisfy the infinitary distributive law:

$$(D) \quad a \wedge \bigvee X = \bigvee\{a \wedge b \mid b \in X\}, \text{ if } \bigvee X \text{ exists.}$$

(The inclusion $\supseteq$ holds in any lattice; for the inclusion $\subseteq$ it is enough to show that $a \wedge c \subseteq \bigvee\{a \wedge b \mid b \in X\}$ for any $c \in X$, due to the properties of $\supset$). Two other important facts are the following.

Fact 4.10 *1. If a complete lattice satisfies the infinitary distributive law (D), it can be turned into a cHa by defining*

$$b \supset c := \bigvee\{d \mid d \wedge b \leq c\}.$$

2. Any Heyting algebra is distributive, i.e. any Ha satisfies

$$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c).$$

For the first statement one has to show that $a \wedge b \leq c \Leftrightarrow a \leq \bigvee\{d \mid d \wedge b \leq c\}$. From left to right is easy; from right to left, notice that if $a \leq \bigvee\{d \mid d \wedge b \leq c\}$, then $a \wedge b \leq b \wedge \bigvee\{d \mid d \wedge b \leq c\}$ and the latter is (by D) equal to $\bigvee\{b \wedge d \mid d \wedge b \leq c\}$, which is just c. The second is easily verified.

We are now ready to give the algebraic semantics for the systems E-PROPn. Let in the following $(A, \bigwedge, \bigvee, \bot, \supset)$ be a cHa. We freely use the notions $\vee$ and $\wedge$, as they were given in Definition 4.9. The interpretation of the terms of E-PROPn will be in A and its higher order function spaces. We therefore let $\lceil - \rceil$ be the mapping that associates the right function space to a domain D, so

$$\begin{array}{rcl} \lceil \mathsf{Prop} \rceil & = & A, \\ \lceil D_1 \to D_2 \rceil & = & \lceil D_1 \rceil \to \lceil D_2 \rceil, \end{array}$$

where the second $\to$ describes function space. In the following we shall freely speak of the 'interpretation of E-PROPn in $(A, \bigwedge, \bigvee, \bot, \supset)$', where of course this interpretation includes the mapping of higher order terms into the appropriate higher order function space based on A.

Definition 4.11 *Let $n \in \mathbb{N} \cup \{\omega\}$. Any cHa Θ is an* algebraic model of E-PROPn.

Note that, as there are no constants in the formal systems of propositional logics, a model does not include a valuation for the constants. (the extension with constants is no problem though.)

Definition 4.12 *The interpretation of* E-PROPn *in the algebraic model $(A, \bigwedge, \bigvee, \bot, \supset)$, $[\![-]\!]$, is defined modulo a valuation ρ for free variables that maps variables of domain D into $\lceil D \rceil$. So let ρ be a valuation. Then $[\![-]\!]_\rho$ is defined inductively as follows.*

$$\begin{array}{rcl} [\![\alpha]\!]_\rho & = & \rho(\alpha), \textit{ for } \alpha \textit{ a variable}, \\ [\![PQ]\!]_\rho & = & [\![P]\!]_\rho [\![Q]\!]_\rho, \\ [\![\lambda x \epsilon D.Q]\!]_\rho & = & \boldsymbol{\lambda} t \in \lceil D \rceil . [\![Q]\!]_{\rho(x:=t)}, \\ [\![\varphi \supset \psi]\!]_\rho & = & [\![\varphi]\!]_\rho \supset [\![\psi]\!]_\rho, \\ [\![\forall x \epsilon D.\varphi]\!]_\rho & = & \bigwedge\{[\![\varphi]\!]_{\rho(x:=t)} \mid t \in \lceil D \rceil\}. \end{array}$$

It is easily seen that $[\![-]\!]_\rho$ satisfies the usual substitution property and that interpretations are stable under $\beta\eta$-equality, i.e.

$$[\![P]\!]_{\rho(x:=[\![Q]\!]_\rho)} = [\![P[Q/x]]\!]_\rho$$

and

$$P =_{\beta\eta} Q \Rightarrow [\![P]\!]_\rho = [\![Q]\!]_\rho.$$

Definition 4.13 *For Γ a finite set of formulas of* E-PROPn*, φ a formula of* E-PROPn *and Θ an algebraic model, φ* is Θ-valid in Γ*, notation $\Gamma \models_\Theta \varphi$, if for all valuations ρ,*

$$\bigwedge\{[\![\psi]\!]_\rho \mid \psi \in \Gamma\} \leq [\![\varphi]\!]_\rho.$$

If Γ is empty we say that φ is Θ-valid *if $\models_\Theta \varphi$.*

Note that $\bigwedge\{[\![\psi]\!]_\rho \mid \psi \in \Gamma\}$ exists, because Γ is finite. In the following we just write $[\![\Gamma]\!]_\rho$ for $\bigwedge\{[\![\psi]\!]_\rho \mid \psi \in \Gamma\}$.

Our definition is a bit different from the one in [Troelstra and Van Dalen 1988], where $\Gamma \models_\Theta \varphi$ is defined by

$$\forall\psi \in \Gamma[[\![\psi]\!]_\rho = \top] \;\Rightarrow\; [\![\varphi]\!]_\rho = \top.$$

Our notion implies the one above, but not the other way around. However, they are the same if $\Gamma = \emptyset$ and they also yield the same consequence relation. One disadvantage of our notion is that we have to restrict to finite Γ. This is easily overcome by putting

$$\Gamma \models_\Theta \varphi \text{ if for all finite } \Gamma' \subseteq \Gamma, \text{ one has } \Gamma' \models_\Theta \varphi.$$

Definition 4.14 *Let Γ be a (finite) set of formulas of* E-PROPn *and φ a formula of* E-PROPn*. We say that φ* is a consequence of Γ*, notation $\Gamma \models \varphi$, if $\Gamma \models_\Theta \varphi$ for all algebraic models Θ.*

Proposition 4.15 (Soundness) *For Γ a finite set of formulas of* E-PROPn *and φ a formula of* E-PROPn*,*

$$\Gamma \vdash_{\text{E-PROP}n} \varphi \Rightarrow \Gamma \models \varphi.$$

Proof Let Θ be a model. By induction on the derivation of $\Gamma \vdash \varphi$ we show that for all valuations ρ, $[\![\Gamma]\!]_\rho \leq [\![\varphi]\!]_\rho$. None of the six cases is difficult. We treat the cases for the last rule being ($\supset$-E) and ($\forall$-I).

($\supset$-E) Say φ has been derived from $\psi \supset \varphi$ and ψ. Let ρ be valuation. Then by IH $[\![\Gamma]\!]_\rho \leq [\![\psi]\!]_\rho$ and $[\![\Gamma]\!]_\rho \leq [\![\psi \supset \varphi]\!]_\rho$. The second implies $[\![\Gamma]\!]_\rho \wedge [\![\psi]\!]_\rho \leq [\![\varphi]\!]_\rho$. So, by $[\![\Gamma]\!]_\rho \leq [\![\psi]\!]_\rho$ we conclude $[\![\Gamma]\!]_\rho \leq [\![\varphi]\!]_\rho$.

($\forall$-I) Say $\varphi \equiv \forall f \epsilon D.\psi$ and $\Gamma' \subseteq \Gamma$ is the finite set of non-discharged formulas of the derivation with conclusion ψ. Then by IH, $\forall\rho[[\![\Gamma']\!]_\rho \leq [\![\psi]\!]_\rho]$, so $\forall\rho\forall F \in \lceil D\rceil[[\![\Gamma']\!]_\rho \leq [\![\psi]\!]_{\rho(f:=F)}]$, because $f \notin \text{FV}(\Gamma')$. This immediately implies that $[\![\Gamma]\!]_\rho \leq [\![\forall f \,\epsilon\, D.\psi]\!]_\rho$. $\Box$

To show completeness we first construct the Lindenbaum algebra for E-PROPn. This is a Ha but not yet a cHa. The construction in [Troelstra and Van Dalen 1988] tells us how to turn it into a cHa which has all the desired properties.

Definition 4.16 *For $n \in \mathbb{N} \cup \{\omega\}$, we define the* Lindenbaum algebra for E-PROPn*, $\mathcal{L}_n$. First we define the equivalence relation $\sim$ on* Sent(E-PROPn) *by*

$$\varphi \sim \psi \;:= \vdash_{\text{E-PROP}n} \varphi \supset \psi \;\&\; \psi \supset \varphi.$$

We denote the equivalence class of φ under $\sim$ by $[\varphi]$. $\mathcal{L}_n$ is now defined as the Ha $(A, \wedge, \vee, \bot, \supset)$ where

$$\begin{aligned} A &= (\text{Sent}(\text{E-PROP}n))_\sim, \\ [\varphi] \wedge [\psi] &= [\varphi \,\&\, \psi], \\ [\varphi] \vee [\psi] &= [\varphi \vee \psi], \\ [\varphi] \supset [\psi] &= [\varphi \supset \psi], \\ [\bot] &= [\bot]. \end{aligned}$$

Note that the &, $\vee$, $\supset$ and $\perp$ on the right of the $=$ are the logical connectives: $\supset$ is basic and the others were defined in Remark 4.2 by

$$\begin{array}{rcl} \varphi \,\&\, \psi & := & \forall \alpha \epsilon \mathsf{Prop}(\varphi \supset \psi \supset \alpha) \supset \alpha, \\ \varphi \vee \psi & := & \forall \alpha \epsilon \mathsf{Prop}(\varphi \supset \alpha) \supset (\psi \supset \alpha) \supset \alpha, \\ \perp & := & \forall \alpha \epsilon \mathsf{Prop} \alpha. \end{array}$$

Each $\mathcal{L}_n$ is obviously a Ha: $[\varphi] \leq [\psi]$ iff $\varphi \vdash_{\text{E-PROP}n} \psi$.

Lemma 4.17 *For Γ a finite set of sentences of* E-PROPn *and φ a sentence of* E-PROPn,

$$\Gamma \vdash_{\text{E-PROP}n} \varphi \;\Leftrightarrow\; \Gamma \leq \varphi \textit{ in } \mathcal{L}_n.$$

Proof Immediate by the construction of $\mathcal{L}_n$. $\square$

Theorem 4.18 ([Troelstra and Van Dalen 1988]) *Each Ha Θ can be embedded into a cHa $c\Theta$ such that $\wedge$, $\vee$, $\perp$, $\supset$ and existing $\bigwedge$ and $\bigvee$ are preserved and $\leq$ is reflected.*

Proof Let $\Theta = (A, \wedge, \vee, \perp, \supset)$ be a Ha. A *complete ideal of* Θ, or just *c-ideal*, is a subset $I \subset A$ that satisfies the following properties.

1. $\perp \in I$,
2. I is *downward closed* (i.e. if $b \in I$ and $a \leq b$, then $a \in I$),
3. I is *closed under existing sups* (i.e. if $X \subset I$ and $\bigvee X$ exists, then $\bigvee X \in I$).

Now define $c\Theta$ to be the lattice of c-ideals, ordered by inclusion. Then $c\Theta$ is a complete lattice that satisfies the infinitary distributive law D, and hence $c\Theta$ is a cHa by defining

$$I \supset J := \bigvee \{K \mid K \wedge I \subset J\}.$$

To verify this note the following.

- $c\Theta$ has infs defined by $\bigwedge_{q\in Q} I_q = \bigcap_{q \in Q} I_q$.
- $c\Theta$ has sups defined by $\bigvee_{q\in Q} I_q = \{\bigvee X \mid X \subset \bigcup_{q\in Q} I_q, \bigvee X \text{ exists}\}$: the set $\{\bigvee X \mid X \subset \bigcup_{q\in Q} I_q, \bigvee X \text{ exists}\}$ is indeed a c-ideal and it is also the least c-ideal containing all I_q.
- $I \cap \bigvee_{q\in Q} I_q = \bigvee \{I \cap I_q \mid q \in Q\}$ and so D holds.

The embedding i from Θ to $c\Theta$ is now defined by

$$i(a) = \{x \in A \mid x \leq a\}.$$

The embedding preserves $\perp$, $\supset$ and all existing $\bigwedge$,

$\bigvee$. For the preserving of $\bigvee$, let $X \subset A$ such that $\bigvee X$ exists in Θ. We have to show that $i(\bigvee X) = \bigvee_{x\in X} i(x)$, i.e. show that

$$\{y \in A \mid y \leq \bigvee X\} = \{\bigvee Y \mid Y \subset \bigcup_{x\in X} i(x), \bigvee Y \text{exists}\}.$$

For the inclusion from left to right, note that $X \subset \{y \in A \mid \exists x \in X[y \leq x]\}$ and so $X \subset \bigcup_{x\in X} i(x)$. This implies that $\bigvee X \in \{\bigvee Y \mid Y \subset \bigcup_{x\in X} i(x), \bigvee Y \text{exists}\}$ and so we are done because the latter is a c-ideal. For the inclusion from right to left, let $z = \bigvee Y_0$ with $Y_0 \subset \bigcup_{x\in X} i(x)$. Then $z \leq \bigvee X$ so we are done.

Finally, the embedding i reflects the ordering, i.e.

$$i(a) \subset i(b) \Rightarrow a \leq b. \square$$

Corollary 4.19 (Completeness) *For* Γ *a finite set of sentences of* E-PROPn *and* φ *a sentence of* E-PROPn*,*

$$\Gamma \models \varphi \Rightarrow \Gamma \vdash_{\text{E-PROP}n} \varphi.$$

Proof Following the Theorem, we embed the Lindenbaum algebra of E-PROPn, $\mathcal{L}_n$, in the algebraic model (cHa) $c\mathcal{L}_n$. This algebraic model $c\mathcal{L}_n$ is complete with respect to the logic. So, for Γ a finite set of sentences and φ a sentence of E-PROPn, we have

$$\Gamma \models \varphi \;\Rightarrow\; \Gamma \models_{c\mathcal{L}_n} \varphi \;\Rightarrow\; \Gamma \leq \varphi \text{ in } \mathcal{L}_n \;\Rightarrow\; \Gamma \vdash_{\text{E-PROP}n} \varphi. \square$$

Corollary 4.20 (Conservativity) *For any* $n \geq 2$, E-PROP$(n{+}1)$ *is conservative over* E-PROPn, *and hence* E-PROPω *is conservative over* E-PROPn.

Proof For Γ a finite set of sentences and φ a sentence of E-PROPn^τ,

$$\Gamma \vdash_{\text{E-PROP}(n+1)} \varphi \Rightarrow \Gamma \models \varphi \Rightarrow \Gamma \vdash_{\text{E-PROP}n} \varphi$$

by soundness and completeness of the algebraic models for any of the E-PROPn.

The conservativity of E-PROPω over E-PROPn is now immediate: any derivation in E-PROPω is a derivation in E-PROPm for some $m \in \mathbb{N}$. $\square$

Corollary 4.21 *For any* $n \in \mathbb{N} \cup \{\omega\}$, PROP$n$ *is conservative over* PROP2.

Proof By the fact that PROPn is a subsystem of E-PROPn and the fact that PROP2 and E-PROP2 are the same system. $\square$

By the fact that the formulas-as-types embedding (from PROPω to $\lambda\omega$, respectively from PROP2 to $\lambda 2$) is an isomorphism, we can immediately conclude the following.

Theorem 4.22 *The type system* $\lambda\omega$ *is conservative over* $\lambda 2$*, that is, for all* $\lambda 2$*-contexts* Γ *and* $\lambda 2$*-types* σ *we have*

$$\Gamma \vdash_{\lambda\omega} M : \sigma \Rightarrow \exists N[\Gamma \vdash_{\lambda 2} N : \sigma].$$

5 Discussion and concluding remarks

5.1 Semantical versus syntactical proofs of conservativity

We have seen that a proof of conservativity between typed λ calculi can be helpful to prove conservativity between logics. An example is the proof of conservativity of λPRED2 over λPRED, which immediately implies the conservativity of second order predicate logic over minimal first order predicate logic. Due to the syntactic nature of the proof (which is done by normalizing the proof terms), it is convenient to use the typed λ calculus format for the conservativity proof. For the proof of conservativity of $\lambda\omega$ over $\lambda 2$, a semantical proof was used. At this point it is not clear to us how a purely syntactical proof can be given. For example, the method of normalizing the proof terms does not work here because of the following.

Fact 5.1 *There are a context* Γ *and a type* A *of* $\lambda 2$ *and a term* M *(of* $\lambda\omega$*) such that*

$$\Gamma \vdash_{\lambda\omega} M : A \text{ and } \Gamma \not\vdash_{\lambda 2} nf(M) : A.$$

One example is found by taking Γ the empty context and A the type of functions from numerals to numerals, so $A \equiv N{\rightarrow}N$, where $N \equiv \Pi\alpha : \star.(A{\rightarrow}A){\rightarrow}A{\rightarrow}A$. Then one can take for M a representation of a recursive function that is not λ-definable in $\lambda 2$. (Such terms exist, due to [Girard 1972], where it is shown that in $\lambda\omega$ more recursive functions are λ-definable then in $\lambda 2$.) Then $\vdash_{\lambda\omega} M : N{\rightarrow}N$, but not $\vdash_{\lambda 2} \mathrm{nf}(M) : N{\rightarrow}N$, because the normal form of M λ-defines the same recursive function as M.

An easier counterexample is found by taking, for example, Γ to be $z : \Pi\alpha{:}\star.\alpha{\rightarrow}\mathrm{True}$ and $A \equiv \mathrm{True}$, where $\mathrm{True} \equiv \Pi\beta{:}\star.\beta{\rightarrow}\beta$. Then $\Gamma \vdash_{\lambda\omega} x(\Pi P{:}\star{\rightarrow}\star.P\,\mathrm{True}) : \mathrm{True}$, which is a term in normal form and not typable in $\lambda 2$.

It would be interesting to see how a syntactic proof of the conservativity of $\lambda\omega$ over $\lambda 2$ could be given. This would give an algorithm that computes for every $\lambda\omega$-term M that has a $\lambda 2$-type A, a $\lambda 2$-term N that also has the type A.

For the proof of conservativity of $\lambda 2$ over $\lambda{\rightarrow}$ (and hence of conservativity of second order propositional logic over minimal first order propositional logic), we used normalization here. This is not really necessary. It is possible to give a semantical proof, using, for example, the algebraic semantics that we discussed for second and higher order propositional logic. It is also possible to define a mapping from propositiona and proofs of second order propositional logic to first order propositional logic that preserves derivability. This is done by Pitts.

One of the consequences of conservativity of PROPω ober PROP2 is that the system PROPω is not decidable. (Both the extensional and the non-extensional version of the system are undecidable.) This follows from the undecidability of PROP2, which was proved by [Löb 1976].

References

[Barendregt 1992] H.P. Barendregt, Typed lambda calculi. In *Handbook of Logic in Computer Science*, eds. Abramski et al., Oxford Univ. Press.

[Barendsen and Geuvers 1989] E. Barendsen and H. Geuvers, λP is conservative over first order predicate logic, Manuscript, Faculty of Mathematics and Computer Science, University of Nijmegen, Netherlands,

[Berardi 1988] S. Berardi, Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Dept. Computer Science, Carnegie-Mellon University and Dipartimento Matematica, Universita di Torino, Italy.

[Berardi 1989] S. Berardi, Talk given at the 'Jumelage meeting on typed lambda calculus', Edinburgh, September 1989.

[Berardi 1990] S. Berardi, Type dependence and constructive mathematics, Ph.D. thesis, Universita di Torino, Italy.

[De Bruijn 1980] N.G. de Bruijn, A survey of the project Automath, In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds. J.P. Seldin, J.R. Hindley, Academic Press, New York, pp 580-606.

[Coquand 1985] Th. Coquand, Une théorie des constructions, Thèse de troisième cycle, Université Paris VII, France, January 1985.

[Coquand 1990] Th. Coquand, Metamathematical investigations of a calculus of constructions. In *Logic and Computer Science*, ed. P.G. Odifreddi, APIC series, vol. 31, Academic Press, pp 91-122.

[Coquand and Huet 1988] Th. Coquand and G. Huet, The calculus of constructions, *Information and Computation*, 76, pp 95-120.

[Coquand and Huet 1985] Th. Coquand and G. Huet, Constructions: a higher order proof system for mechanizing mathematics. *Proceedings of EUROCAL '85, Linz*, LNCS 203.

[Van Daalen 1973] D. van Daalen, A description of AUTOMATH and some aspects of its language theory, In P. Braffort, ed. *Proceedings of the symposium on APL, Paris.*

[Geuvers 1989] J.H. Geuvers, Talk given at the 'Jumelage meeting on typed lambda calculus', Edinburgh, September 1989.

[Geuvers and Nederhof 1991] J.H. Geuvers and M.J. Nederhof, A modular proof of strong normalisation for the calculus of constructions. *Journal of Functional Programming*, vol 1 (2), pp 155-189.

[Geuvers 1993] J.H. Geuvers, Logics and Type systems, PhD. Thesis, University of Nijmegen, Netherlands.

[Girard 1972] J.-Y. Girard, Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur. Ph.D. thesis, Université Paris VII, France.

[Girard 1986] J.-Y. Girard, The system F of variable types, fifteen years later. *TCS* 45, pp 159-192.

[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.

[Harper et al. 1987] R. Harper, F. Honsell and G. Plotkin, A framework for defining logics. *Proceedings Second Symposium on Logic in Computer Science*, (Ithaca, N.Y.), IEEE, Washington DC, pp 194-204.

[Howard 1980] W.A. Howard, The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, eds. J.P. Seldin, J.R. Hindley, Academic Press, New York, pp 479-490.

[Longo and Moggi 1988] G. Longo and E. Moggi, Constructive Natural Deduction and its "Modest" Interpretation. Report CMU-CS-88-131.

[Paulin 1989] Ch. Paulin-Mohring, Extraction des programmes dans le calcul des constructions, Thèse, Université Paris VII, France.

[Ruys 1991] M. Ruys, $\lambda P\omega$ is not conservative over $\lambda P2$, Master's thesis, University of Nijmegen, Netherlands, November 1991.

[Tonino anf Fujita 1992] H. Tonino and K.-E. Fujita, On the adequacy of representing higher order intuitionistic logic as a pure type system, *Annals of Pure and Applied Logic* 57, pp 251–276.

[Löb 1976] M. Löb, Embedding first order predicate logic in fragments of intuitionistic logic, *J. Symbolic Logic* vol 41, 4, pp. 705–719.

[Troelstra and Van Dalen 1988] A. Troelstra and D. van Dalen, *Constructivism in mathematics, an introduction, Volume I/II*, Studies in logic and the foundations of mathematics, vol 121 and volume 123, North-Holland.

[Verschuren 1990] E. Verschuren, Conservativity in Barendregt's cube, Master's thesis, University of Nijmegen, Netherlands, December 1990.

Logic of refinement types

Susumu Hayashi*
Department of Applied Mathematics and Informatics
Ryukoku University
Ohtsu, Shiga, JAPAN

Aug 1993

Abstract

ATTT is a conservative extension of Girard-Reynold's second order polymorphic lambda calculus $\lambda 2$ introduced in [8]. ATTT has refinement types which are intended to be subsets of ordinary types or specifications of programs. In this paper, we will study the logic of refinement types. (Warning: logic of refinement types is irrelevant to refinement logic in the sense of R. Constable.)

1 Introduction

ATTT is a conservative extension of Girard-Reynold's second order polymorphic lambda calculus $\lambda 2$ (or F in more popular name). The aim of ATTT is to serve yet another type theoretic basis for program extraction or program development via type theories. In [8], the theory of pure ATTT was presented and its basic metatheory was developed. In this paper, we will study logics in ATTT. The main purpose of this paper is to show how logic is represented by refinements of ATTT. We will show that the refinements of each type is an "internally" complete Heyting algebra (cHa). Using the cHa structure, second order intuitionistic logic is naturally interpreted by refinements using the technique of categorical logic [14]. The second order formulas provable in the intuitionistic second order logic are provaly true in ATTT under this interpretation. Conversely, the first order formulas provaly true in ATTT under this interpretation are provable in the intuitionistic first order logic. We conjecture that this holds for the second order case as well, and will give a sufficient condition.

The logic of refinements mentioned above is "non-informative" in the sense of Coq system [3] or "rank zero" in the sense of PX system [7]. It is used as a substitute of logics of Coq's propositions and PX's rank zero formulas. This means that the interpretation of formulas does keep any computational information contrary to the ordinary BHK-interpretation (Brouwer-Heyting-Kleene-interpretation) of constructive logic. Besides this non-informative interpretation, Some BHK-style informative interpretations of logic can be coded in ATTT. They are essentialy realizability interpretations. We will show that both of recursive realizability and modefied realizability are represented naturally in ATTT. Then, we will show that informative induction principle is derivable by non-informative induction principle in ATTT.

We also introduce a notion of refinement classfier resembling suboject classifier of topos, and, by means of this notion, we will compare our apprach to program/proof development in ATTT with related works [2], [3], [15], [18].

*hayashi@rins.st.ryukoku.ac.jp

2 Refinement types

In this section, we will briefly review the notion of refinement types and develope cHa-structures of the refinements.

2.1 Types versus refinement types

A type is a collection of data which have the same uniform structure. A set is a collection of data which have the same property. Property which defines a set can be so complicated that we cannot effectively decide if a data belongs to the set. On the other hand, types (of programming languages) are supposed to be simple. For examples, integers, elements of free algebras, records and functions are all types. But, the primes are not a type, but a set. We summerize this intuition by the following doctorine: the border of types are smooth and the border of sets are rugged.[1] "The function f returns the next prime number to its real argument" is a specification of a function f. The edge of this specification is rugged, as it uses the primes in it.

Intuitively, sepcifications must be identified with sets. Thus, we think that the specifications should be distinguished from data types, even in the type theories for program extraction based on Curry-Howard isomorphism. This consideration led us to introduce a new kind of "types" called *refinement types*. Refinement types have rugged borders and are intended to serve as specifications. This refinement types "refine" the borders of types, i.e., they distinguish elements more refinedly than types. More precisely, a refinement types of type A is intended to be a *subset* of the ordinary type A. Note taht a refinement type, or refinement in short, is not a subtype of A, since it may have a rugged border.

ATTT is a type system with refinement types designed as "ATTT = $\lambda 2$ + refinement types." In a sense , ATTT is "$\lambda 2$ + logic." In set theory, sets are defined by the aid of logical formulas. We do not introduce formulas, but can directly define set (refinements) by means of the rules for refinements in the framework of type theory. In categorical or algebraic logic, logics are coded through sets (subobject), e.g. the statement "$F(x)$ implies $G(x) \vee H(x)$" can be interpreted as set-theoretic inclusion $\{x|F(x)\} \subset \{x|G(x)\} \cup \{x|H(x)\}$. Thus we may think that the sets part (refinements part) of ATTT is a kind of logic *without* formulas. This resembles the philosophy of Martin-Löf's type theory (at least his philosphy in 80's). The difference is that Martin-Löf used Curry-Howard isomorphism to represent logic but we use more "traditional" approach via categorical or algebraic logic. Note that we have *not* abandoned Martin-Löf's philosophy. The categorical-algebraic logic approach is used only for non-informative part. For informative part, we follow his philosophy. In a sense, we use Curry-Howard isomorphism, when we sit in outside of Martin-Löf's subset type, but use categorical-algebraic logic approach, when we sit in inside of Martin-Löf's subset type.

More formally, ATTT is described as follows. The types and terms of ATTT are exactly the ones of $\lambda 2$. If A is a type of $\lambda 2$ (and so of ATTT), we introduce the refinement kind of A, which we will write $\mathit{refine}(A)$. A refinement kind is a sort of "kind" in the sense of GTS. The elements of the refinement kind $\mathit{refine}(A)$ is intended to be the refinements of the type A. Each refinement kind $\mathit{refine}(A)$ is closed under the formation rules of singleton $\{a\}_A$ ($a \in A$), finite or infinite union and intersection. Furthermore, if R_1 is in $\mathit{refine}(A)$ and R_2 is in $\mathit{refine}(B)$, then $R_1 \to R_2$ is in $\mathit{refine}(A \to B)$. This function type like refinement represents the constructive implication. Natural introduction and elimination rules for these refinements are included. A subtle point is the introduction rule for the refinement $R_1 \to R_2$. If $e \in R_2$ for $x \in R_1$, then the

[1] I learnt this "doctorine" from Rod Burstall. According him, it's due to JÅṘobinson.

term $\lambda x \in A.e$ is introduced in $R_1 \to R_2$, and not $\lambda x \in R_1.e$.[2] The description of ATTT above would be enough to understand the development below. Thus we do not enter the details more. (See [8] for more details.)

2.2 Refinement application

For the discussions below, we will introduce a notion *refinement application*. A future of ATTT is that the realizability relation "a realizes F" can be a refinement not only a judgment. The relation is expressed as

$$F \wedge \{a\}.$$

This refinement is realized by a, if and only if, a belongs to F. If we consider a refinement of A as a predicate on A as we later do, $F \wedge \{a\}$ is considered the application of F to a. So we call this refinement application and write $F\{a\}$.

Note that Martin-Löf's propositional equality $I(F, a, a)$ is a proposition which expresses realizability relation. The refinement application resemble this. The difference is $F\{a\}$ is realized by a, and $I(F, a, a)$ is realized by a fixed constant.

2.3 The algebra of refinements

The refinement kind $\mathit{refine}(A)$ is intended to be the subsets of the type A. So it has a lattice theoretic structure. $\bigwedge$ and $\bigvee$ are "internal" infimum and supremum, respectively. Actually, they are an "internally" complete Heyting algebra. Namely, they are distributed "internally" complete lattice. To show it, we define an order relation on $\mathit{refine}(A$. The relation itself is a refinements of $\mathit{refine}(A \to A)$.

Definition 1 (subset refinement Let R_1 and R_2 be refinements of a type A. The *subset refinement* $R_1 \leq R_2$ is defined as follows:

$$R_1 \leq R_2 \stackrel{\text{def}}{=} \bigwedge x \in R_1.R_1\{x\} \to R_2\{x\}.$$

The subset refinement is a refinement of type $A \to A$. Intuitively, the subset refinement says that R_1 is a subset of R_2.

For this refinement the following proposition holds.

Proposition 1 *Let $A \in \mathit{Type}, R_1 \in \mathit{refine}(A), R_2 \in \mathit{refine}(A)$ be derivable under a context Γ. Then, the following are equivalent:*

1. *$\Gamma, x \in R_1 \vdash x \in R_2$ is derivable.*
2. *$\Gamma \vdash \lambda x \in A.x \in R_1 \leq R_2$ is derivable.*
3. *$\Gamma \vdash e \in R_1 \leq R_2$ is derivable for some term e.*

[2] This rule is justified by the observation that terms of ATTT are exactly those of $\lambda 2$ and the former lambda term is a term of $\lambda 2$ but the latter is not. This rule also reflects the "typing" rule of modified realizabilities. Recently, Pfenning introduced a type theory with intersection types which has a function-introduction rule similar to ours for a different good reason [17].

Note that the second condition shows that the identity function is the standard witness of the refinement. In this sense, the subset refinement is self-realizing or non-informative.

It is easy to see the subset refinement defines an pre-order relation on $\mathit{refine}(A)$. Note that this fact can be stated *internally*. Namely, the followings are derivable in an appropriate context:

1. $\bigwedge R \in \mathit{refine}(A).R \leq R$,
2. $\bigwedge R_1 \in \mathit{refine}(A). \bigwedge R_2 \in \mathit{refine}(A).\ \bigwedge R_3 \in \mathit{refine}(A).R_1 \leq R_2 \supset R_2 \leq R_3 \supset R_1 \leq R_3$.

Note that "$\supset$" is the ordinary implication defined by the function space. (Since the subset refinement is self-realizing, the implication may be replaced by non-informative implication explained later.)

Since the subset relation is a pre-order, we have to have the "equivalence relation" induced from the pre-order. This can be also internalized. Let's call such an equivalence relation *extensional equality refinement*, $R_1 \geq\leq R_2$ in notation. It's defined by

$$R_1 \geq\leq R_2 \stackrel{\text{def}}{=} R_1 \leq R_2 \leq R_2 \geq\leq R_1.$$

2.4 The refinements are internally complete Heyting algebra

The finite intersection and union are infimum and maximum, respectively. These facts can be stated internally as well as externally. As we have big intersections and unions, the refinements of a type are *internally* complete. For example, we can derive the followings for the big intersection:

$$\ldots, i \in I \vdash \lambda x \in A.x \in (\bigwedge i \in I.R) \leq R$$

$$\frac{\ldots, X \in \mathit{refine}(A), i \in I \vdash e \in X \leq R}{\ldots, X \in \mathit{refine}(A) \vdash X \leq \bigwedge x \in A.R}(i \text{ is not free in } e)$$

Note that we have to be careful with what kind of indexes $i \in I$ are allowed. It depends on what kind of formation rules for the intersection and union are allowed. But, once a formation rule is introduced, there is no difficulty to prove the properties above.

A complete Heyting algebra is a complete lattice with the following distribution law:

$$a \wedge \bigvee i \in I.b_i = \bigvee i \in I.(a \wedge b_i).$$

The distribution law is provable in ATTT.

Lemma 1 (the distribution lemma) *Assume i is not free in S. In the contexts where R and S are refinements of a type A, the following refinement is provably inhabited.*

$$R \wedge \bigvee i \in I.S \geq\leq \bigvee i \in I.(R \wedge S).$$

Note that I must be an index set (in the context) for which the formation of the big intersection allowed.

This lemma is not quite trivial. The $\geq$-direction is trivial, but the $\leq$-direction is not trivial. In some type theories with intersection and union types, the distribution law is included as an axiom, as the $\leq$-direction is not derivable from the other axioms and rules. In ATTT, the $\leq$-direction is easily derivable by the following lemma:

Lemma 2 *The following are derived rules of ATTT:*

1.

$$\frac{\Gamma \vdash e_0 \in \bigvee x \in A.B}{\Gamma \vdash e_0 \in \bigvee x \in A.(B \wedge \{e_0\})}$$

2.

$$\frac{\Gamma \vdash e_0 \in \bigvee x \in A.B \qquad \Gamma, x \in A, y \in B \wedge \{e_0\} \vdash e_1 \in C}{\Gamma \vdash e_1[y := e_0] \in C[y := e_0]},$$

where x is not in $FV(e_1) \cup FV(C)$.

The second one is a consequence of the first one.

2.5 implication refinement

In cHa, implication $a \supset b$ is defined as the greatest element c such that $a \wedge c \leq b$ holds. By the same vain, we define the *implication refinement.*

Definition 2 (implication refinement) Let R_1 and R_2 be refinements of a type A. Then we define the *implication refinement* $R_1 \check{\supset} R_2$ by

$$\bigvee R \in \mathit{refine}(A). \bigvee w \in R_1 \wedge R \leq R_2.R.$$

Note that the implication refinement is of type A and the subset refinement is of type $A \rightarrow A$. For the implication refinement, the following two are equivalent:

$$\Gamma, a \in R_1 \vdash a \in R_2,$$

$$\Gamma, a \in A \vdash a \in R_1 \check{\supset} R_2.$$

This implies the equivalence of the followings:

$$\Gamma, a \in R_1\{a\} \vdash a \in R_2\{a\},$$

$$\Gamma, a \in A \vdash a \in (R_1 \check{\supset} R_2)\{a\}.$$

By the commutativity, we can prove that $(R_1 \check{\supset} R_2) \wedge \{a\}$ and $(R_1 \wedge \{a\} \check{\supset} R_2 \wedge \{a\}$ are extensionally equal refinements. If we abuse the notation, we have

$$\Gamma, R_1\{a\} \vdash R_2\{a\},$$

$$\Gamma, a \in A \vdash R_1\{a\} \check{\supset} R_2\{a\}.$$

This shows that the implication refinement satisfies the natural deduction rules for the familiar implication.

2.6 Absurdity refinement and truth refinement

The absurdity refinement of a type A, $\perp_A$, is the least refinement and the negation refinement is define from it and the implication.

$$\perp_A \stackrel{\text{def}}{=} \bigwedge R \in \mathit{refine}(A).R,$$

$$\check{\neg} R \stackrel{\text{def}}{=} R \check{\supset} \perp_A.$$

Obviously, the absurdity refinement implies any refinement of the same type. The truth refinement $\top_A$, which is the greatest refinement is also definable by

$$\top_A \stackrel{\text{def}}{=} \bigvee R \in \mathit{refine}(A).R.$$

Normally, Heyting algebra is assumed to have at least two elements. But, we do not assume in this paper. As a type may be empty in a semantics for $\lambda 2$, we cannot prove the truth refinement is not the same as the absurdity refinement.

Note that $\perp$ is not unique in ATTT. We have the absurdity $\perp$ for each refinement kind. Later, we will show the refinement of the unit type *unit* classifies the refinements. Thus, $\perp_{unit}$ is the most general absurdity in a sense. If $x \in A$ holds, then $R\{x\}$ for any refinement R of A under the assumption $\perp_{unit}$.

Proposition 2 *The following is derivable in ATTT:*

$$\lambda A \in \mathit{Type}.\lambda x \in A.x \in \Pi A \in \mathit{Type}.\bigwedge R \in \mathit{refine}(A).\bigwedge w \in \perp_{unit}.A \leq R.$$

This means that any refinement is true, if we assume $\perp_{unit}$.

3 Interpreting logic by refinements

In this section, we give an interpretation of second order logic by refinements. A refinement kind of a type A is the power set of A. Thus, refinement kinds resemble the algebras of subobjects in a category. We give an interpretation of second order predicate logic into ATTT using the technique of categorical logic by Makkai and Reyes [14]. But, our case is some different from theirs. We will consider second order logic, on the other hand, Makkai and Reyes considered first order logic. They used external intersections and unions are used, but we will use internal ones. Furthermore, products are assumed in [14]. We define product by the polymorphic product (see [8]). The polymorphic products are not real products but semi-products in the sense of [6] and [9], since $\lambda 2$part of ATTT does not have η-conversion.[3]

3.1 The Makkai-Reyes-style interpretation

In spite of the differences mentioned above, Makkai-Reyes technique works well in our case by small modifications. Let's illustrate the interpretation briefly. Let D be a fixed type. This is intended to be the domain of first order objects (ground type) for the second order logic that we are going to interpret. Let F be second order formulas whose *first order* free variables are among $x_1, ldos, x_n$. Their interpretations are refinements of the type D^n. F may have second order free variables. Thus, the interpretation of these must be given. We assign refinements with appropriate types to them. The interpretation of F is defined depending on this assignment.

[3] Note that we can define a refinement $\bigvee a \in A. \bigvee b \in B.\{\langle a, b\rangle\}$ of a polymorphic product $A \times B$, which is the "real" product. But, this trick would be "incorrect," as a product of types should be a type not a refinement.

3.1.1 Interpretation of predicates

Let's see how to interpret predicates by refinements. Since refinements are sets, they are unary predicates. There are no refinements to represent predicates with many arguments. Thus, we use the polymorphic products and paring to interpret n-ary predicates. For example, a binary predicate $P(x, y)$ is interpreted as a refinement R of the polymorphic product $D \times D$. More precisely, we assign a $R \in \mathit{refine}(D \times D$ to P, and interpret the formula $P(x, y)$ by

$$\{\langle x, y\rangle \in D \times D | R\{\langle x, y\rangle\}\}.$$

The pair $\langle x, y\rangle$ is the polymorphic Church pairing and the subset notation is an abbreviation of

$$\bigvee x \in D. \bigvee y \in D. \bigvee w \in R\{\langle x, y\rangle\}.\{\langle x, y\rangle\}_{D\times D}.$$

(See [8].)

3.1.2 Interpretation of logical operators

Conjunction, disjunction, implication and negation of formulas are interpreted as intersection, union, implication and negation of refinements. Note that implication is interpreted by implication refinement defined above. This is straightforward, but, there is a problem.

If F and G have different free variables, this interpretation does not work, since their interpretations are of refinements of different types. Then, we embed refinements into a bigger refinements. Assume we are to form the union of the interpretations of F and G, say R_F and R_G. If F has a free variable x_1 as $F(x_1)$ and G has another free variable x_2 as $G(x_1, x_2)$. Then, $R_F \in \mathit{refine}(D)$ and $R_G \in \mathit{refine}(D \times D)$. Thus, we have to embed the interpretation of $F(x_1)$ into $D \times D$ as $\{\langle x_1, x_2\rangle in D \times D | R_F\{x_1\}\}$.

Now, we interpret the universal and existential quantifiers. This is the point at which our interpretation diverges from Makkai-Reyes interpretation. Makkai and Reyes interpret first order quantifiers by images and "co-images" of subobject morphisms. We interpret them by the big intersection and union over the type D. Let $F(x_1, \ldots, x_n)$ be a formula and let R_F be its interpretation, which is a refinement of D^{n+1}. The interpretation of the formula $\forall x_i.F(x_1, \ldots, x_n)$, is the refinement

$$\bigwedge x_i \in D.\{\langle x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\rangle \in D^n | R\{\langle x_1, \ldots, x_n\rangle\}\}.$$

First order existential quantifier is interpreted in the similar way by union.

Since we have unions and intersections over refinement kinds, second order quantifiers are interpreted in the same way. For example, let P be a binary predicate variable. Then $forallP.F$ is interpreted by $\bigwedge P \in \mathit{refine}(D^2).R_F$. Although P runs through the refinements of the polymorphic product D^2, which may have more elements than the real pairs, this is the same to have it run through the refinements of the real products. This is because that P always appears in the form $P\{\langle s, t\rangle\}$ in F. This completes the definition of the interpretation of logic by refinements.

3.2 Provably true formulas in ATTT

A refinement R of a type A is said to be true, if and only if $\lambda x \in A.x \in \top_A \leq R$ holds. [4] If this judgment is provable in ATTT, we will say the refinement R is provaly true in ATTT. If the

[4]More exactly speaking, we have to respect in which context the refinement is true.

interpretation of a formula F is provably true in ATTT, then we say the formula is provably true in ATTT.

3.2.1 Intuitionistic second order logic are provably true

Then, the following is easily proved:

Proposition 3 *The intuitionistic second order (predicate) logic (ISOL) is sound w.r.t. the interpretation. Namely, if a closed formula F of ISOL is provable, then it is provably true in ATTT under the context $D \in Type, x \in D$, where the type variable D is used as the domain of first order objects.*

The intuitionistic predicate logic assumes that the domain is non-empty. Thus, the assumption $x \in D$ is necessary. If we use a free logic instead, this is not necessary.

3.2.2 What are provably true formulas?

It is natural to ask what are provably true formulas in ATTT? We conjecture that provably true formulas in ATTT are exactly the formulas provable in ISOL. This means that our interpretation is adequate to ISOL, and amounts to the following:

Conjecture 1 *The converse of proposition 3 holds. Namely, if a closed formula F of ISOL is provably true in ATTT under the context mentioned in proposition 3, F is provable in ISOL.*

Somehow related result can be found in [13]. Although, we have not been able to prove this conjecture yet, we will prove the conjecture restricted to first order formulas. Namely, the following proposition holds:

Proposition 4 *If a closed* first order *formula F is provably true in ATTT under the context mentioned in the proposition above, then F is provable in the intuitionistic* first order *predicate logic (IFOL).*

The idea of our proof is very simple. But, full details are rather clumsy. Thus, we illustrate only the idea, below.

Our proof consists three steps. Firstly, we show that any provaly true first order formula F is valid in the sense of Tarski semantics by means of a term model of $\lambda 2$. Secondly, we formalize the proof of this fact in the intuitionistic second order arithmetic (HAS) augumented with an auxiliary sort γ. The auxiliary sort is intended as the generic domain, i.e., we do not pose any axiom or rules for this except the ones of ISOL. Let us call this system HAS(γ). By the formalization, we can show that F^γ, which is F whose unique sort is made to be γ, is porvable in HAS(γ). Lastly, we show that if F^γ is provable in HAS(γ) then F is provable in ISOL. Thus, F is provable in ISOL.

The first step is easy. If the judgment maintaining that the interpretation is true is provable in ATTT under the judgment, we can interpret it in the set-theoretical sematics introduced of [8]. Let *Const* be a arbitrary set of new constans. We augment $\lambda 2$ by *Const* as a new type. Namely, we introduce a new type constant D and regard the constants from *Const* as new constants of $\lambda 2$. Let denote such the augumented system $\lambda 2(D)$. Let $\mathbf{M}$ be the *closed* term model of $\lambda 2(D)$.

In [8], we took BMM-model as the basis of semantics of ATTT. But, $\mathbf{M}$ is not a BMM-model but a $\lambda 2$-algebra in the sense of [10]. This is not important for us. Any concrete $\lambda 2$-algebra also gives a semantics of ATTT in the sense of [8]. The reason why we considered only BMM-model in [8] is only for simplicity. The semantics works for any other "concrete" semantics of $\lambda 2$, e.g., coherent semantics etc.

The model $\mathbf{M}$ consists of the collection of the set of closed normal terms of $\lambda 2(D)$. We denote the set of closed normal terms of type A by $\mathbf{M}(A)$. By interpreting *refine*(A) as the power set of the closed normal terms of type A, it becomes a model of ATTT (see [8]). Assume that a first order formula F is provaly true in ATTT, interpreting D is the domain of the first order objects. Since the normal closed terms of D are only the new constants, $\mathbf{M}(D)$ is identical to *Const*. Let $x_1, \ldots, x_n$ be the free variables of F. Then, it is easy to check that R_F (the interpretation of F) coincides with the set of the list of constants of *Const* $\{\langle c_1, \ldots, c_n \rangle$, where $F[c_1/x_1, \ldots, c_n/x_n]$ is valid in the sense of Tarski semantics whose domain is *Const*. Since *Const* is arbitrary, this means that F is tautology in the sense of Tarski semantics. Thus, F is provable in the classical first order logic by the completeness theorem. Note that the argument above is all constructive except the completeness theorem.

The second step involves messy formalization. Let us clarify the system HAS(γ)in which we do the formalization. HAS$(\gamma)$is a second order theory with two sorts, ν and γ. The sort ν is intended to be the sort of natural numbers and the sort γ is intended to be arbitrary. The constants of ν is only 0, and γ does not have any constants. The only function symbol is s (successor). The arity of a predicate variable is a list $(s_1, \ldots, s_n$ of sorts (n is possibly zero). Thus we can talk about functions from ν to γ, etc. The equality for each sort is defined by Leibniz equality. (This is only for simplicity. We may have equality symbols, instead.) The logic is the (two-sorted) intuitionistic second order logic, and the axioms are the Peano axiom for the sort of natural numbers ν.

Now, we formalize the argument of the first step in HAS(γ). The only pointwise formalization is possible, since we need normalization theorem of $\lambda 2$. We augment ATTT with D just as $\lambda 2(D)$. Let us denote the augmented system by ATTT(D). Assume that a first order formula F is provaly true in ATTT. Then it is also provably true in ATTT(D), regarding the new type D as the domain of the first order objects. Then there is a fragment of $\lambda 2(D)$ for which the strong normalization property is provable in HAS(γ) and all terms in the proof of the truth of F fall in the fragment (see, e.g., [19]). Thus, we have a closed term model of the fragment formalized in HAS(γ). Since the set of constants *Const* is arbitrary, we may regard it as the sort γ. Recall that the Makkai-Reyes-style interpretation coincided with the Tarski semantics. Thus, we can conclude that F^D is provable in HAS(γ), where F^D is F replaced the ground sort by γ.

The third step consists of the following lemma:

Lemma 3 *A first order formula F is provable in intuitionistic first order logic, if F^D is provable in HAS(γ).*

This is an easy consequence of infinitary normalization theorem of intuitionistic second order arithmetic which is proved by Tait-Girard computability predicate technique, e.g., [5]. A proof of F^D of HAS(γ) can be normalized using ω-rule. Since F^D is a first order formula only with the sort γ, subformula property holds, namely, the sort appearing in the normal proof is only γ. This means that the proof is a proof of ISOL and, in fact, of the intuitionistic first order logic, regarding the sort γ as the ground sort. This completes the proof.

The only third step fails for the second order case. Since second order logic does not have subformula property, our proof fails to prove the lemma above for second order case. But, we conjecture that the lemma will hold for second order formulas as well.

Conjecture 2 *A second order formula F is provable in intuitionistic second order logic, if F^D is provable in HAS(γ).*

Note that this lemma has an intrinsic meaning: if a second order formula is constructively valid under the assumption of the existence of the natural numbers, then it is provable in second order intuitionistic logic. Since this seems very plausible, the conjecture must be true.

3.3 The excluded middle

The law of excluded middle for refinements

$$\Pi A \in \mathit{Type}. \bigwedge R \in \mathit{refine}(A).(\top_A \leq R \vee \dot{\neg} R)$$

is not derivable in ATTT by the the adequacy result above. Thus, the Heyting algebra of a refinement kind is not boolean. But, they are compatible with boolean laws. Even if we add the law of excluded middle above as an axiom, the system is still conservative over $\lambda 2$. In [8], we defined a translation from ATTT to $\lambda 2$ which is identical on the $\lambda 2$ fragment of ATTT. We may add the excluded middle above as an axiom, whose realizer is the identity function. Then the translated axiom is $\lambda x \in A.x \in A \rightarrow A$.

Note that the excluded middle inequivalent to the following:

$$\Pi A \in \mathit{Type}. \bigwedge R \in \mathit{refine}(A).(\not{R} \leq R).$$

This resembles the logic of PX system, which is compatible with classical logic as stressed in [7].

3.4 The refinement classifier and comparisons to the other approaches

The Makkai-Reyes-style interpretation given above look rather different from the standard way of interpreting logic in higher order type systems as Coq system. Thus, it seems difficult to compare our approach to the standard ones. But, we can encode logic similarly to the standard way by the notion of refinement classifier. Upon the idea, we can compare our approach to the others.

3.4.1 The refinement classifier

Refinements resembles subojects in toposes. A topos has a subobject classfieirs by which subobjects are classified. Namely, a subobject of an object is represented by a map from the object to the subobject classifier. Intuitively, a subobject classfier is the power set of a singleton set. It can be considered as the set of truth values. Introducing a singleton type, say *unit*, consisting the only element $\star$, the refinement kind $\mathit{refine}(\mathit{unit})$ is a refinement classifier. Note that this is the kind of non-informative propositions, and so corresponds to *Prop* in the sense of Coq system rather than the *Prop* in the sense of the original Calculus of Constructions.

We will write the refinement kind $\mathit{refine}(\mathit{unit})$ by Ω and call the refinement classifier. Assume that $R \in \mathit{refine}(A)$. Then we define a term of the type $A \rightarrow \Omega$. Then the following correspondence between $A \rightarrow \Omega$ and $\mathit{refine}(A)$ is obtained:

$$R \in \mathit{refine}(A) \quad \longmapsto \quad \lambda x \in A. \bigvee w \in R\{x\}.\{\star\},$$
$$F \in A \in \Omega \quad \longmapsto \quad \bigvee x \in A. \bigvee w \in F(x)\{\star\}.\{x\}.$$

But, $A \to \Omega$ is illegal in ATTT in [8]. Thus, we have to extend ATTT to include such a type by allowing the following formation rule and corresponding introduction rule.

$$\frac{\Gamma \vdash A \in \mathit{Type} \qquad \Gamma \vdash B \in \mathit{RKind}}{\Gamma \vdash A \to B \in \mathit{RKind}}$$

This extension is not essential. In such a extension, any term of *RKind* has the form $A_1 \to \ldots A_n \to \mathit{refine}(B)$, where $A_1, \ldots, A_n$ are types. Then, we can regard it as $\mathit{refine}(A_1 \times \ldots \times A_n \times B)$, by regarding $\lambda x_1 \in A_1. \cdots \lambda x_n \in A_n.R$ as $\{\langle x_1, \ldots x_n \rangle \in A_1 \times \cdots A_n \times B | R\}$. Note that this is a standard techinque to encode functions form first order objects to predicates in the second order logic. Thus, the extension still keep the second order character.

3.4.2 Comparisons to the other approach

Upon the notion of refinement classfier, we compare our approach to the others. In impredicative higher order type theories as Calculus of Constructions, *Type* (or often denoted as *Prop*) is often used as a substitue of the collection of the truth values. It is also regarded as the collection of the data types. This confusion leads to inefficiency of the extracted code and makes it difficult to understand encoded logic. Thus, Coq system separate these two notions introducing two kinds *Prop* and *Set*. The kinds *Prop* and *Set* belong to the sorts (in the sense of GTS) *Type* and *Type_Set*, respectively. There is a messy confusion of symbols. *Prop* of Coq corresponds to our Ω and *Set* of Coq corresponds to our *Type*, the kind of the types. *Type* of ATTT belongs to the sort *Kind* and refinement kinds $\mathit{refine}(A)$ belong to the sort *RKind*. Thus, *Type* of Coq is our *Kind* and *Type_Set* is our *RKind*.

The superficial divergences of our approach from Coq's are

1. . ATTT is a second order system, but Coq is a higher order system.
2. . The types of Coq can depend on data, as it uses Calculus of Construction. The types of ATTT cannot depend on data, as it uses $\lambda 2$.

There is still another twist. Coq has an extraction algorithm, by which the data-dependency of types (and other logical information) is stripped out so that the extracted codes are of Girard' F_ω. At the specification level, Coq has data-dependent types, but, at the object code level, it does not have them. The types of ATTT are the ones of $\lambda 2$ at the both levels. In ATTT, we directly talk about object level codes and their types, i.e. the terms and types of $\lambda 2$. On the other hand, the users of Coq talk about them indirectly through the extraction procedure. This is fate of the systems based on realizability as Coq and PX. We do not say this is a bad doom, as the indirectness can make users think their programs abstractly and good extraction procedure can sometime produce better codes than the users expected. This contrast between direct and indirect approaches may be compared with the contrast between assembler languages and compiler langauges.

Another approach closely related to Coq is "deliverable" by Burstall and McKinna ([2], [15]). They take a direct approach. They do not consider any extraction algorithm, but directly talk about codes. To represent data types and talk about them and codes, they use Luo's Extended Calculus of Constructions ECC ([11], [12]). ECC has *Prop* and a predictive cumulative hierarchy of predictive kinds, $Type(0)$, $Type(1), \ldots$. They use *Prop* as the truth values and $Type(i)$ as data types. Thus, *Prop* corresponds to our Ω. But, there is a difference. *Prop* belongs to and is a subset of $Type(0)$. This implies that the collection of truth values and each truth value is a data type. It would be possible to regard the collection of truth values as a data type by regarding them as formulas. But, a proposition (truth value) is a refinement of a unit type from our point of view. Thus, it has a rugged border, and so it is not a data type. When it comes to data-dependency of types, deliverable approach allows it. The predictive hierarchy allows such types.

The approach most closed to ours is Erik Poll's programming logic $\lambda\omega_L$ [18]. As programming language, he uses Girard's F_ω, $\lambda\omega$ in his notation. As logic, he uses the other copy of $\lambda\omega$. His $*_s$ is *Type* and $*_p$ is Ω of ATTT. These two copies of the same type theory resembles two copies of Calculus of Constructions in the Coq system. Then he introduce function types from a data type A to $*_p$ to represent predicates. This corresponds to the refinement kind $refine(A)$ or $A \to \Omega$. The data-dependency of types is not allowed as he uses F_ω. In a sense, $\lambda\omega_L$ is a higher order version of ATTT without refinements. The difference is that Poll uses predicates, on the other hand, we use sets (refinement types). As propositions and data types, and so programs and proofs, are more clearly separated in $\lambda\omega_L$ than ECC, $\lambda\omega_L$ might be a better framework for the deliverable approach.

As illustrated above, the four approaches are closely related in a rather intricate way. They do not seem very different from a theoretical point of view. The difference in practice of proof/program developments should be investigated.

4 Informative interpretations of logic

The logic of refinements given above is a non-informative interpretation. In this section, we examine two informative interpretations in ATTT.

4.1 Two informative interpretations of implication

Type theories based on "proofs as program" notion can be regarded as "axiomatizations" of realizability notions. For example, Martin-Löf's type theory can be regarded as an axiomatization of extensional recursive realizability as exploited in [1].

In PX, realizers of non-informative formulas are always the empty list which are passed by extracted codes. In Coq, realizers of non-informative formulas are the null sequence which literally disappear in extracted codes. These two different treatments of non-informativeness are reflected by the difference of realizability interpretations used by these systems. PX uses recursive realizability and Coq uses modified realizability.

The difference of these realizabilities appears in the interpretation of implication. Let P be a non-informative formula and let A be an informative formula. In recursive realizability,

$$r \text{ realizes } P \supset A \stackrel{\text{def}}{=} r\downarrow \wedge P \supset r(nil) \text{ realizes } A.$$

In modefied realizability,

$$r \text{ realizes } P \supset A \stackrel{\text{def}}{=} r\downarrow \wedge P \supset r \text{ realizes } A.$$

(Normally, only terminating terms are used in modefied realizability. Then, the condition $r\downarrow$ may be omitted.) In ATTT, user can choose both intepretations locally. Let R_1 and R_2 be refinements. The standard intuitionistic implication is defined by $R_1 \to R_2$. This interpretation takes R_1 informative.

The following interpretations take R_1 non-informative:

1. $\bigwedge w \in R_1.R_2$,
2. $\{x \in \mathit{unit}|R_1\} \to R_2$,

w is a fresh variable which does not appear in R_2. The first one represents the implication of the modefied realizability, and the second one represents the implication of the recursive realizability. In this sense, ATTT embodies both realizabilities in a single framework.

4.2 Induction principles

Induction principle is a very important componet of a system of constructive programming, since it is the device to derive recusive programs. Below, we will examine two kinds of induction principles, one is non-informative and the other is the informative.

The non-informative one is purly logical. The informative one is not only logical but also representing recursion scheme as well. For example, informative mathematical induction represents primitive recursion and its correctness. We will show that informative induction principle can be derived from non-informative induction principle in ATTT. This is a future which distinguishes ATTT from the other type theories.

4.3 Monotone inductive definition of refinements

Firstly, we will examine non-informative induction principles. They are induction principles over refinements. The monotone inductive definition of refinements are easily defined as in the second order logic. Let R be a refinement variable of type A and let $\Psi(R)$ be a refinement of A. Then, $\mu R.\Psi(R)$ is

$$\bigwedge S \in \mathit{refine}(A). \bigwedge w \in (\Psi(S) \leq S).S.$$

If $\Psi(S) \leq S$ holds, then, obvioulsy, $\mu R.\Psi(R) \leq S$ holds. If Φ is monotone, i.e., $R_1 \leq R_2 \supset \Phi(R_1) \leq \Phi(R_2)$, holds, then $\Phi(\mu R.\Psi(R)) \geq\leq \mu R.\Psi(R)$ holds. In this sense, $\mu R.\Psi(R)$ is the least fixed point of a monotone operator $\Phi(R)$.

4.4 The type of natural numbers

Recursive data types as integers, list, etc. are definable in $\lambda 2$. For example, natural numbers are definable as Church integers. But, we do not use this as data types. We should introduce such types as primitive type as in [3]. For simplicity, we consider only Nat, here.

To extend ATTT by Nat, we introduce a type constant $\mathit{Nat}$and two constants $0 \in \mathit{Nat}$ and $S \in \mathit{Nat} \to \mathit{Nat}$. Then, we add the primitive recursor natrec

$$\mathit{natrec} \in \Pi X \in \mathit{Type}.X \to (\mathit{Nat} \to X \to X) \to \mathit{Nat} \to X$$

together with the standard reductions.

To say Nat is built by 0 and S, we introduce the following axiom:

$$\mathit{Nat} \ \leq \mu R.\{x\}_{\mathit{Nat}} \ \vee S''R,$$

where $S''R$ is the image of R by S, which is defined by

$$f''R \stackrel{\mathrm{def}}{=} \bigvee x \in R.\{f(x)\}.$$

Since *Nat* is not a refinement, "$Nat \ \leq \mu R. \ldots$" is not the order on refinements. This is defined by $\bigwedge x \in Nat.\{x\}_{Nat} \to (\mu R. \ldots)\{x\}$.

As the axiom is self-realizing by the identity function, the axiom may be put in the context with a variable, which is a virtual witness of the axiom. These are all of the axioms for *Nat*.

Note that we did not introduce the informative mathematical induction. The last axiom is a kind of mathematical induction, but it is non-informative mathematical induction (NIMI). The informative mathematical induction (IMI) should be

$$\Pi A \in \mathit{Type}. \bigwedge P \in Nat \to \Omega.P(0) \to \Pi n \in Nat.(P(n) \to P(Sn)) \to \Pi n \in Nat.P(n).$$

In ATTT, IMI is provable from the axioms above.

Theorem 1 *The following is derivable in ATTT:*

$$\lambda A \in \mathit{Type}.\lambda a \in Nat.\lambda f \in Nat \to A \to A.\lambda n \in Nat.recAafn \in \mathit{IMI}.$$

Note that we have to put the abstractions, since ATTT does not have η-conversion. Note that this theorem asserts that the big λ-term above is the realizar of IMI. Let $e(n)$ be the term $recAafn$. It is enough to prove $e(n) \in P(n)$ under the assumptions. To prove this, it is sufficient to prove the refinement $P(n)\{e(n)\}$. We can prove this by the non-informative mathematical induction NIMI. The proof is essentially the same as the soundness proof of mathematical induction for realizabilities.

This example shows that informative induction follows from non-informative induction in ATTT. This ability of ATTT seems that distinguish it from the other type theories and illustrates its set theoretical nature.

There is one thing missing in the axioms above. It is Peano's fourth axiom "$0 = 1$ implies $\perp$." There are several formulations of the axiom, since we have several ways to define implication and absurdity. One of the strongest formulation would be

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1.\perp_A.$$

The type of this refinement is $PiA \in \mathit{Type}.A$. This is rather problematic as pointed out in [16]. If the refinement is realized by f, then $f(A)$ is a polymorphic element of A. In PX system, this problem did not arise, as it is a monotype system whose domain is inhabited. But, we are now in a type theoretic framework and so cannot use such a solution. In Coq system [16], "abort"-command is introduced which belongs polymorphically to any type A. This solution seems to lead us to a type theory with non-terminating programs. Coq system offers a solution, but it is not very clear.

This approach shows a resembles to the resent works on controls in Curry-Howard isomorphisms by Griffin, Murthy, Nakano. Especially, Nakano's calculus accommodate catch/throw mechanism in a natural constructive framework which have termination property for *all* types. (In Griffin and Murthy's work, the termination seems restricted to specific types.) As "abort" is a throw to the top-level, there might be a natural way to interpret this problem by Nakano's calculus.

Anyway, the problem does not seem to have been settled satisfactory. Thus, we do not use it, here. Here, we consider a less ambitious formulation,

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1. A \to \bot_A,$$

which is equivalent to

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1. \bot_{unit}.$$

In Coq system, essentially the same statement is proved by a proof by cases, which is similar to Martin-Löf's proof of the 4th axiom in his type theory. If we introduce a principle to define refinement-valued functions by cases, we can do the same thing in ATTT. Assume that we can define the following program from Nat to Ω.

$$\lambda x \in Nat.(\mathit{if}\ \ x = 0\ \ \mathit{then}\ \top_{unit}\ \mathit{else}\ \bot_{unit}).$$

By assuming $0 = 1$, we can convert top_unit to $\bot_{unit}$. This proves the axiom above by means of conversion rule or an appropriate equality rule. Note that the program above have the type $Nat \to \mathit{refine}(unit)$. If we extend the type of the recursor rec so that it permits refinement kinds as well as the type A, then we can do such a definition by cases. This doesn't seem bad. But we do not do so here. We simply add the axiom to the system.

Note that type A is assumed to be non-empty, when we deduce bot_A:

$$\Pi A \in \mathit{Type}. \bigwedge w \in 0 = 1. A \to \bot_A.$$

(This means that "assume we have an element of A. If something is wrong, we return it as the default value.") The non-emptiness assumption would not cause a big problem in practical cases. Such an A would be the type of a specification. A specification of a program of a type whose elements are not known will be meaningless.

Acknowledgments

I am very grateful to Rod Burstall and Henk Barendregt for inviting me to the BRA workshop "Types for Proofs and Programs." I thank Christine Paulin, Erik Poll, Kees Hemerik, Benjamin Werner, Mariko Yasugi for discussions on the subjects of the paper and/or Coq system which helped me to clarify the nature of ATTT.

References

[1] M.J. Beeson, *Foundations of Constructive Mathematics*, Springer-Verlag, 1985.

[2] R.M. Burstall and J. H. McKinna, *Deliverables: an approach to program in Constructions*, Technical Report ECS-LFCS-91-133, Department of Computer Science, The University of Edinburgh, 1991.

[3] Dowek, C. et al., *The Coq Proof Assistant User's Guide*, Version 5.6, Technical Report No. 134, INRIA, December, 1991.

[4] Th. Coquand and G. Huet, Calculus of Constructions, *Information and Computation*, vol 76, pp. 95-120, 1988.

[5] S. Hayashi, On derived rules of intuitionistic second order arithmetic, *Commentariorum Mathematicorum Universitatis Sancti Pauli*, vol. XXVI, pp. 77-103, 1977.

[6] S. Hayashi, Adjunction of semifunctors: categorical structures in non-extensional lambda calculus, *Theoretical Computer Sciences,*, vol. 41, pp.95-104, 1986.

[7] S. Hayashi and H. Nakano, *PX: A Computational Logic*, The MIT Press, 1988.

[8] S. Hayashi, *Singleton, Union and Intersection Types for Program Extraction*, Lecture Notes in Comuter Science No. 526, pp. 701-730, T.Ito and A.R.Meyer, eds., Springer-Verlag, 1991, an extended version to appear in Information and Computation.

[9] R. Hoofman, The theory of semi-functors, *Mathematical Structures in Computer Science*, vol. 3, pp.93-128, 1993.

[10] B. Jacobs, Semantics of the second order lambda calculus, *Mathematical Structures in Computer Science*, vol. 1, pp. 327-360, 1991.

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

[12] Z. Luo. *An Extended Calculus of Constructions*, Ph. D. thesis, Department of Computer Science, The University of Edinburgh, 1990.

[13] Z. Luo. *A problem of adequacy: conservativity of Calculus of Constructions over higher order logic*, Technical Report ECS-LFCS-90-121, Department of Computer Science, The University of Edinburgh, 1990.

[14] M. Makkai and G. Reyes. *First Order Categorical Logic*, Lecture Note in Mathematics No. 611, 1977, Springer-Verlag.

[15] J. H. McKinna. *Deliverables: A Categorical Approach to Program Development in Type Theory*, Ph. D. thesis, Department of Computer Science, The University of Edinburgh, 1992.

[16] C. Paulin-Mohring and B. Werner, Synthesis of ML programs in the System Coq, Journal of Symbolic Computation, 1993.

[17] F. Pfenning, Intersection Types for a Logical Framework, in this volume, 1992.

[18] Erik Poll, A programming logic for F_ω, Computing Science Notes, 92/25, Departement of Mathematics and Computing Science, Eindhoven University of Technology, 1992.

[19] A.S. Troelstra, *Metamathematical investigations of intuitionistic arithmetic and analysis*, Lecture Notes in Mathematics, vol. 344, Springer-Verlag, 1973.

Proof-Checking a Data Link Protocol

L. Helmink[¶]
Philips Research Laboratories

M.P.A. Sellink[‖]
Utrecht University

F.W. Vaandrager[**]
CWI and University of Amsterdam

September 1993

Abstract

A data link protocol developed and used by Philips is modeled and verified using I/O automata theory. Correctness is computer-checked with the Coq proof development system. The protocol does not assume fairness of data transmission channels and message-length is not restricted.

Key words: I/O Automata, Proof-Checking, Type Theory.

The work of the first author is partially supported by ESPRIT Basic Research Action 6453: 'Types for Proofs and Programs'. The work of the third author took place in the context of the ESPRIT Basic Research Action 7166: 'CONCUR2'.

> Upon completion of this article, the authors repaired a deadlock in an earlier version of the automaton description of the protocol (Section 3.2). The deadlock was not revealed until Lemma 3.16 was proof-checked. The refinement proof including the invariants have been adapted and verified by hand, but re-checking all invariants could not be completed in time.

1 Introduction

The data-link layer of a telecommunication protocol is verified and proof-checked. The protocol has been designed to communicate large messages over unreliable channels. The messages are transmitted in small packets or *frames*. The protocol does not rely on fairness of data transmission channels, i.e., repeated transmission of a frame does not guarantee its eventual arrival. For this reason, the number of retransmission attempts is limited and the protocol is called Bounded Retransmission Protocol.

[¶]helmink@prl.philips.nl
[‖]alex@phil.ruu.nl
[**]fritsv@cwi.nl

Reliable communication protocols are vital to the telecommunication industry. They are also of increasing importance to the electronics business because more and more products consist of communicating subsystems and because many products integrate technology from the fields of computers, telecommunication devices, and consumer electronics. The pressure for reliability of the protocols involved poses an important challenge to verification techniques.

Design, implementation and testing of communication protocols is a complicated and error-prone activity. For many protocol-based products, erroneous protocol behavior is met by error-recovery procedures or by issueing a new software release. For some products however, error situations are not acceptable and software maintenance is impossible. Correctness of protocols is usually examined by careful testing of implementations.

Thorough testing increases confidence but testing is only semi-decidable: it may reveal the presence of errors but not the absence of errors. Protocol verification is required to obtain a higher degree of confidence. The protocol is modeled in a mathematical structure and correctness is guaranteed by showing that the protocol satisfies the required behavior under all circumstances. Verification is not restricted to implementations but can also be applied to designs that have not yet been implemented. It should be stressed however that although verification excludes design errors, it cannot replace testing of implementations.

A hand-written protocol verification may itself contain certain errors that can be eliminated by computer tools. Verification errors can be classified into two types: wrong assumptions and wrong deductions, corresponding to errors in the protocol model and to errors in its correctness proof, respectively. Errors of the first type are the responsibility of the modeler. Errors of the second type can be eliminated using computer tools for proof development or proof-checking. There is an additional advantage to the use of computer tools in protocol verification. Protocol verification is a labour-intensive and a non-trivial activity: much effort of skilled experts is required. With the current state-of-the-art, it is cost-effective only for those (parts of) protocols that are truly critical. Computer tools will enable more efficient verification of protocols.

In this paper we describe a verification and the associated proof-checking for a data-link protocol. First, the protocol is proven correct using the input/output automaton model of Lynch and Tuttle [13], a formalism based on extended finite state machines. Next, the verification is proof-checked in type theory with the Coq system [7]. The protocol is a simplified and stylized version of a Philips telecommunication protocol. The objective of this work is twofold. The primary objective is to prove correctness of the protocol with the highest possible level of confidence. The second goal of this work is to bring to light all technical issues that are involved in obtaining this result.

A starting point for the work described here was an algebraic specification of the protocol in PSF [17], a language based on process algebra. This specification was developed and validated using PSF simulation tools. The PSF description was translated into IO-automata theory and a suitable correctness criterion was defined. The protocol was verified by proving that it satisfies the correctness criterion. This specification and verification were then translated into type theory and checked with the Coq proof development system.

This paper is divided into the following parts: Section 2 gives an informal description of the protocol. Next, Section 3 explains the verification of the protocol. Section 4 discusses the proof-checking with the Coq system. Section 5 concludes with a discussion of the results.

2 Protocol Outline

Like most data link protocols, the Bounded Retransmission Protocol can be regarded as an extended version of the Alternating Bit Protocol. The protocol uses a stop-and-wait approach known as 'positive acknowledgement with retransmission' [21]: after transmission of a frame the sender waits for an acknowledgement before sending a new frame. The protocol procedures are similar to the LAPB link control procedures of the X.25 protocol [22] (for X.25 acknowledged mode and window size = 1, viz. one outstanding unacknowledged frame). Incoming frames are checked for errors. Correctly received frames are acknowledged while erroneous frames are simply discarded. If the acknowledgement fails to appear, the sender times out and retransmits the frame. An alternating bit is used to detect duplication of a frame. Real-time aspects are limited to the use of time-outs to detect loss of frames and loss of acknowledgements. Three service primitives are offered by the protocol: a request and confirm service at the sender side, and an indication service at the receiver side.

- $REQ(s)$
  The request service to transmit a finite list s of data. Each datum corresponds to a message frame.

- $CONF(c)$ $\quad (c \in \{\mathsf{C\_OK}, \mathsf{C\_NOT\_OK}, \mathsf{C\_DONT\_KNOW}\})$
  The confirmation service that informs the sender about the result of a request.

  - $c = \mathsf{C\_OK}$: the request has been dispatched successfully.
  - $c = \mathsf{C\_NOT\_OK}$: the request has not been dispatched of completely.
  - $c = \mathsf{C\_DONT\_KNOW}$: the request may or may not have been handled completely. This situation occurs when the last frame is not acknowledged.

- $IND(d, i)$ $\quad$ (d a datum and $i \in$\{$\mathsf{I\_FIRST}$, $\mathsf{I\_INCOMPLETE}$, $\mathsf{I\_OK}$\})
  The indication service to pass a new frame to the receiver.

  - $i = \mathsf{I\_FIRST}$:
    the packet is the first one of a message; more data to follow.
  - $i = \mathsf{I\_INCOMPLETE}$:
    the packet is an intermediate one; more data to follow.
  - $i = \mathsf{I\_OK}$:
    the packet is the last one of a series, completing the transmission of a message.

- IND_NOT_OK
  The indication service to report loss of contact to the receiver. Only part of a message has been received.

The protocol control procedures will be described by means of a sender S, a receiver R, and two communication channels K and L (Figure 1). We will assume that K and L are lossy channels: message frames are either lost or they arrive without corruption in the order in which they are sent. Messages can be communicated over ports *REQ, CONF, F, G, A, B, IND*. A data frame consists of a datum preceded by a header with three information bits named *first*, *last* and *toggle*: $F(\mathit{first}, \mathit{last}, \mathit{toggle}, \mathit{datum})$. *first* and *last* indicate if a packet is the first or last frame of a series, respectively. For a single-frame message both are set. *toggle* plays the role of alternating bit to distinguish between subsequent data frames. Acknowledgement frames consist of these three information bits only: $A(\mathit{first}, \mathit{last}, \mathit{toggle})$.

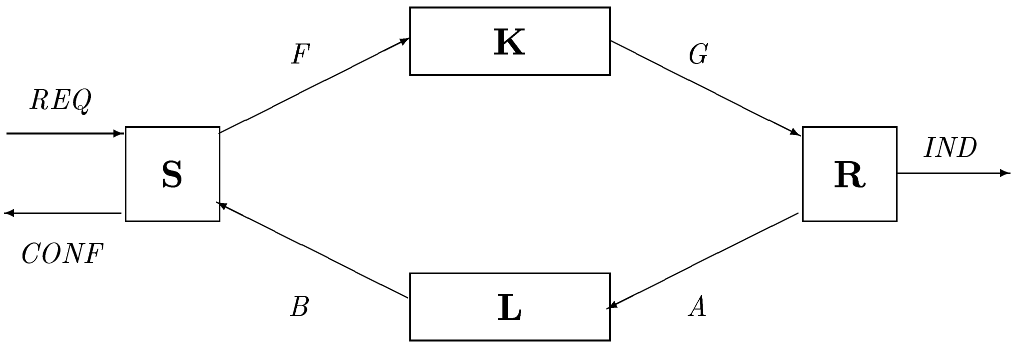


Figure 1: Bounded Retransmission Protocol.

First consider a faultless transmission where no frames are lost. The sender S receives a request to transmit data $d_1 \ldots d_n$: $\mathit{REQ}(d_1 \ldots d_n)$ (Here we will assume n> 2; case n=1 or n=2 are similar). A frame $F(\mathsf{true}, \mathsf{false}, \mathit{toggle}, d_1)$ is sent on port F. Channel K passes on the frame to receiver R over port G. R then issues an $\mathit{IND}(d_1, \mathsf{I\_FIRST})$ to port IND, and sends an acknowledgement frame $A(\mathsf{true}, \mathsf{false}, \mathit{toggle})$ on port A, which is passed on by channel L to port B. The acknowledgement frame consists of the header of the data frame. Upon receipt of the acknowledgement, the sender transmits the second datum: $F(\mathsf{false}, \mathsf{false}, \neg \mathit{toggle}, d_2)$, where *toggle* has flipped. The receiver issues $\mathit{IND}(d_2, \mathsf{I\_INCOMPLETE})$ and acknowledges the frame: $A(\mathsf{false}, \mathsf{false}, \mathit{toggle})$. This procedure is repeated until the last frame is sent with first=false, last=true, and $\mathit{datum}=d_n$. The receiver sends $\mathit{IND}(d_n, \mathsf{I\_OK})$ to report completion of the message and acknowledges receipt. The sender then informs the application of the successful dispatch of the transmission request with $\mathit{CONF}(\mathsf{C\_OK})$.

Now consider the loss of data frames or acknowledgement frames. First consider the loss of frames from the sender point of view. Upon transmission of a frame the sender starts a timer t_1 and waits until either the frame is acknowledged or the timer goes off. If the acknowledgement is received, the timer is switched off and the next frame is sent. The timer is attuned to exceed the round trip time for sending a data frame and receipt of its acknowledgement. If the timer goes off no acknowledgement can come anymore and the frame is retransmitted.

The number of retransmission attempts is bounded by a parameter max, and if this maximum number of retransmissions has been reached, the sender gives up. The confirmation service is invoked in one of two ways: if the data frame in question is not the last frame of a series, then $\mathit{CONF}(\mathsf{C\_NOT\_OK})$ confirms failure of message transfer. For the last data frame, a $\mathit{CONF}(\mathsf{C\_DONT\_KNOW})$ is called: there is no way the sender can tell if the last frame was lost and never arrived or if its acknowledgement was lost.

Finally consider the loss of frames from the receiver point of view. Suppose a lost data frame is not the first one, i.e. the receiver is expecting a data frame follow-up. Upon receipt of a data frame, the receiver starts a timer t_2 and goes to a waiting state. When a data frame arrives it is acknowledged and timer t_2 is switched off. If the data frame has a flipped *toggle* then it is new and it is also indicated to the upper layers. When no data frame arrives, timer t_2 goes off eventually and service $\mathit{IND\_NOT\_OK}$ is called. It is easy to see that timer $t_2 > n{*}t_1$.

3 Verification

3.1 I/O Automata Theory

In this section we give a brief account of those parts of I/O automata theory that we need for the purposes of the paper. For an introduction to the I/O automata model we refer to [13, 14].

3.1.1 I/O automata

An *action signature* S is a triple $(in(S), out(S), int(S))$ of three disjoint sets of respectively *input actions*, *output actions* and *internal actions*. The derived sets of *external actions*, *locally controlled actions* and *actions* of S are defined respectively by

$$\begin{aligned} ext(S) &= in(S) \cup out(S), \\ local(S) &= out(S) \cup int(S), \\ acts(S) &= in(S) \cup out(S) \cup int(S). \end{aligned}$$

An *input/output automaton* A (also called an *I/O automaton*) consists of five components:

- an action signature $sig(A)$,
- a set $states(A)$ of *states*,
- a nonempty set $start(A) \subseteq states(A)$ of start states,
- a set $steps(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$ of *transitions*, with the property that for every state s and input action a in $in(sig(A))$ there is a transition (s, a, s') in $steps(A)$, and
- an equivalence relation $part(A)$ on $local(sig(A))$, having at most countably many equivalence classes.

We let s, s', u, u',.. range over states, and a,.. over actions. We write $s \xrightarrow{a}_A s'$, or just $s \xrightarrow{a} s'$ if A is clear from the context, as a shorthand for $(s, a, s') \in steps(A)$. Also, we will write $in(A)$ for $in(sig(A))$, $out(A)$ for $out(sig(A))$, etc.

An action a is said to be *enabled* in a state s, if $s \xrightarrow{a} s'$ for some s'. Since every input action is enabled in every state, I/O automata are said to be *input enabled*. The intuition behind the input-enabling condition is that input actions are under control of the environment, and that the system that is modeled by an I/O automaton cannot prevent the environment from doing these actions. The partition $part(A)$ describes, what intuitively are the 'components' of the system, and will be used to define fairness.

3.1.2 Composition

A finite collection $S_1, \ldots, S_n$ of action signatures is *strongly compatible* if, for all $i, j \in \{1, \ldots, n\}$ satisfying $i \neq j$, $out(S_i) \cap out(S_j) = \emptyset$ and $int(S_i) \cap acts(S_j) = \emptyset$. We say that a collection of I/O automata are *strongly compatible* if their action signatures are strongly compatible.

The *composition* $S = \prod_{i=1}^{n} S_i$ of a finite collection of strongly compatible action signatures $S_1, \ldots, S_n$ is defined to be the action signature with

- $in(S) = \bigcup_{i=1}^{n} in(S_i) - \bigcup_{i=1}^{n} out(S_i)$,

- $out(S) = \bigcup_{i=1}^{n} out(S_i)$,
- $int(S) = \bigcup_{i=1}^{n} int(S_i)$.

The *composition* $A = \|_{i=1}^{n} A_i$ of a finite collection of strongly compatible I/O automata $A_1, \ldots, A_n$ is the I/O automaton defined as follows:

- $sig(A) = \prod_{i=1}^{n} sig(A_i)$,
- $states(A) = states(A_1) \times \cdots \times states(A_n)$,
- $start(A) = start(A_1) \times \cdots \times start(A_n)$,
- $steps(A)$ is the set of triples $(\vec{s}, a, \vec{s'})$ in $states(A) \times acts(A) \times states(A)$ such that, for all $1 \leq i \leq n$, if $a \in acts(A_i)$ then $\vec{s}[i] \xrightarrow{a}_{A_i} \vec{s'}[i]$, and if $a \notin acts(A_i)$ then $\vec{s}[i] = \vec{s'}[i]$.
- $part(A) = \bigcup_{i=1}^{n} part(A_i)$.

Notice that A is an I/O automaton indeed: $start(A)$ is nonempty because all the sets $start(A_i)$ are nonempty, A is input enabled because all the automata A_i are input enabled, and $part(A)$ is a partition of $local(A)$. We will sometimes write $A_1 \| \cdots \| A_n$ for $\|_{i=1}^{n} A_i$.

3.1.3 Hiding

If S is an action signature and $I \subseteq out(S)$, then the action signature $\mathsf{HIDE}\ I\ \mathsf{IN}\ S$ is defined as the triple $(in(S), out(S) - I, int(S) \cup I)$. If A is an I/O automaton and $I \subseteq out(A)$, then $\mathsf{HIDE}\ I\ \mathsf{IN}\ A$ is the I/O automaton obtained from A by replacing $sig(A)$ by $\mathsf{HIDE}\ I\ \mathsf{IN}\ sig(A)$, and leaving all the other components unchanged.

3.1.4 Traces and fair traces

Let A be an I/O automaton. An *execution fragment* of A is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ of states and actions of A, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* of A is an execution fragment that begins with a start state. We denote by $execs(A)$ the set of executions of A. A state s of A is *reachable* if it is the final state of some finite execution of A.

Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of A. Then the *trace* of α is the subsequence of $a_1 a_2 \cdots$ consisting of the external actions of A. With $traces(A)$ we denote the set of traces of executions of A. For s, s' states of A and β a finite sequence of external actions of A, we define $s \stackrel{\beta}{\Longrightarrow}_A s'$ iff A has a finite execution fragment with first state s, last state s' and trace β.

A *fair execution* of an I/O automaton A is defined to be an execution α of A such that the following conditions hold for each class C of $part(A)$:

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many occurrences of actions from C, or α contains infinitely many occurrences of states in which no action from C is enabled.

This says that a fair execution gives fair turns to each class of $part(A)$, and therefore to each component of the system being modeled. A state of A is said to be *quiescent* if only input actions are enabled in this state. Intuitively, in a quiescent state the system is waiting for an input from the environment. A finite execution is fair if and only if the final state of this execution is quiescent. We denote the set of fair executions of A by $fairexecs(A)$, and the set of traces of fair executions of A by $fairtraces(A)$. Also, we write $qexecs(A)$ for the set of finite fair executions of A, and $qtraces(A)$ for the traces of finite fair executions of A.

3.1.5 Implementation

In I/O automata theory, inclusion of fair traces is commonly used as implementation relation, i.e., we say that an I/O automaton A *implements* an I/O automaton B if $fairtraces(A) \subseteq fairtraces(B)$.

3.1.6 Refinements

In the literature, a whole menagerie of so-called simulation techniques has been proposed to prove that the set of (fair) traces of one automaton is included in that of another. We refer to [16] for an overview and for further references. In this paper we only need a very simple type of simulation, which is called *weak refinement.*

Suppose A and B are I/O automata with the same input and output actions. A *weak refinement* from A to B is a function r from $states(A)$ to $states(B)$ that satisfies the following two conditions:

1. If $s \in start(A)$ then $r(s) \in start(B)$.
2. If s is a reachable state of A and $s \xrightarrow{a}_A s'$, then $r(s) \stackrel{\beta}{\Longrightarrow}_B r(s')$ where β equals a if $a \in ext(A)$, and is empty otherwise.

Theorem 3.1 *If there exists a weak refinement from A to B, then $traces(A) \subseteq traces(B)$.*

Proof: Straightforward from the definitions. ■

The reverse implication does not hold, i.e. there exist I/O automata A and B such that $traces(A) \subseteq traces(B)$, but no weak refinement from A to B can be given. In those cases one has to use other, more general simulations. Also, if there exists a weak refinement from A to B then it is not in general the case that A implements B. However, in the protocol that we analyze in this paper we will establish a weak refinement that maps fair executions to fair executions, and this additional property immediately implies inclusion of fair traces.

3.1.7 The precondition/effect style

In the I/O automata approach, the automata that model the basic building blocks of a system are usually specified in the so-called *precondition/effect* style. For the description of automata one assumes a many-sorted signature Σ together with a Σ-algebra $\mathcal{A}$ which gives meaning to the function and constant symbols in Σ. To describe properties, we use a first-order language over signature Σ with equality and inequality predicates, and the usual logical connectives ($\mathsf{true}, \mathsf{false}, \neg, \wedge, \vee, \rightarrow, \mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ ., \exists, \ldots$).

An *I/O automaton generator* G consists of the following components:

- three pairwise disjoint finite sets $in(G)$, $out(G)$ and $int(G)$ of *action types*, i.e. expressions of the form $a : \mathbf{S}_1 \times \cdots \times \mathbf{S}_n$ with $\mathbf{S}_1, \ldots, \mathbf{S}_n$ sorts of Σ,
- a finite set $states(G) = \{x_1, \ldots, x_m\}$ of (sorted) variables,
- a formula $start(G)$, in which the variables from $states(G)$ may occur free,
- a finite set $steps(G)$ of *transition types*, which are expressions of the form

  $a(y_1, \ldots, y_n)$
  Precondition:
  b
  Effect:
  $x_1 := e_1$
  $\vdots$
  $x_m := e_m$

  such that there is an action type $a : \mathbf{S}_1 \times \cdots \times \mathbf{S}_n$ in $acts(G)$ with each variables y_i of sort $\mathbf{S}_i$, b is a formula, in which variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ may occur free, and which is true if $a : \mathbf{S}_1 \times \cdots \times \mathbf{S}_n$ is in $in(G)$, and the e_j are expressions with the same sort as x_j, in which the variables $x_1, \ldots, x_m, y_1, \ldots, y_n$ may occur,
- an equivalence relation $part(A)$ on $local(G)$.

Each I/O automaton generator G denotes an I/O automaton A in the obvious way: for each action type $a : \mathbf{S}_1 \times \cdots \times \mathbf{S}_n$ and for each choice of values $v_1, \ldots, v_n$ taken from the domains of $\mathbf{S}_1, \ldots, \mathbf{S}_n$, respectively, we introduce an action $a(v_1, \ldots, v_n)$ of A. States of A are interpretations of the variables of $states(G)$ in their domains. Start states of A are those states that satisfy formula $start(G)$. There is a transition

$$s \xrightarrow{a(v_1,\ldots,v_n)}_A s'$$

iff there is some transition type of the above form such that, if ξ is a valuation that agrees with s on $states(G)$ and maps, for $1 \leq i \leq m$, variable y_i to v_i, b evaluates to true under ξ, and, for $1 \leq j \leq n$, e_j evaluates to $s'(x_j)$ under ξ. Finally, $part(G)$ trivially induces a partition on $local(A)$.

For each transition type t of the above form, we define the formula $enabled(t)$ by

$$enabled(t) \quad \triangleq \quad \exists y_1, \ldots, y_n \,.\, b$$

Let $L(G) \triangleq \{t_1, \ldots, t_k\}$ be the set of transition types for locally controlled actions of G. We define the formula $quiescent(G)$ by

$$quiescent(G) \quad \triangleq \quad \bigwedge_{t \in L(G)} \neg enabled(t)$$

Then it follows that a state s of the automaton associated to G is quiescent iff it satisfies formula $quiescent(G)$.

The reader will observe that the translation from I/O automata generators to I/O automata is quite straightforward, and that these two notions are very similar. In fact, Lynch and Tuttle

[13, 14] do not even bother to distinguish between these two levels of description. For the formalization of I/O automata theory in Coq the distinction between the "semantic" level I/O automata and the "syntactic" level of I/O automata generators is of course important, which is why we have discussed it here. The definition of I/O automata generators has been inspired by similar definitions in the work of Jonsson (see, for instance, [11]). In the sequel we will, like Lynch and Tuttle, often refer to I/O automata when we actually mean I/O automata generators.

3.2 Protocol Specification

In this section, we present the formal specification of the Bounded Retransmission Protocol. Following a brief description of the many-sorted algebra that we use, we will first give I/O automata for each of the components of the protocol, and then define the full protocol as the parallel composition of these I/O automata. At the end of this section we will moreover present the definition of an I/O automaton that gives the intended external behavior of the protocol. Since the *BRP* protocol has been explained already in considerable detail in Section 2, we will not repeat that explanation here, and confine ourselves in this section to the formal definitions, together with a brief discussion of some of the notation and certain modeling assumptions.

3.2.1 Data types

We start the specification of the Bounded Retransmission Protocol with a description of the various data types that we will need. We assume a many-sorted signature Σ and a Σ-algebra $\mathcal{A}$ which consist of the following components:

- a sort **Bool** of booleans with constant symbols true and false, and a standard repertoire of function symbols ($\wedge$, $\vee$, $\neg$, $\rightarrow$), all with the standard interpretation over the booleans. Also, we need, for all sorts **S** of Σ, equality and inequality function symbols, and an if-then-else function symbol, all with the usual interpretation:

$$\begin{array}{rcl} .=. & : & \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ .\neq. & : & \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ \mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ . & : & \mathbf{Bool}\times\mathbf{S}\times\mathbf{S} \rightarrow \mathbf{S} \end{array}$$

  Note the (harmless) overloading of the constants and function symbols of sort **Bool** with the propositional connectives used in formulas. We will frequently view boolean valued expressions as formulas, i.e., we use b as an abbreviation of $b{=}\mathsf{true}$.

- a sort **Nat** of natural numbers, with constant symbol 0, successor function symbol succ, and function symbol $\leq : \mathbf{Nat}\times\mathbf{Nat} \rightarrow \mathbf{Bool}$, all with the usual interpretation. We also need a constant symbol max, which denotes the maximum number of retransmissions within the protocol.

- a sort **Data** of data elements that the protocol has to transmit. We find it convenient to assume the presence of a constant symbol $\perp$ of sort **Data**, which denotes the *undefined* data element.

- a sort **List**, with as domain the collection of finite lists over the domain of **Data**. There is a constant symbol ϵ, denoting the empty list, and a function symbol $\mathsf{add} : \mathbf{Data}\times\mathbf{List} \rightarrow$

List, denoting the operation of prefixing a list with a data element. Besides these two constructors, there are function symbols

$$\begin{array}{rcl} \mathsf{hd} & : & \textbf{List} \to \textbf{Data} \\ \mathsf{tl} & : & \textbf{List} \to \textbf{List} \\ \mathsf{one} & : & \textbf{List} \to \textbf{Bool} \end{array}$$

hd denotes the operation of taking the first element of a list, tl denotes the operation that returns the remainder of a list after removal of the first element, and one denotes the operation that returns true iff the argument list has length one. These operations are fully characterized by the axioms (where s is a variable of sort **List**, and d, e are variables of sort **Data**):

$$\begin{array}{rcl} \mathsf{hd}(\epsilon) & = & \bot \\ \mathsf{hd}(\mathsf{add}(d, s)) & = & d \\ \mathsf{tl}(\epsilon) & = & \epsilon \\ \mathsf{tl}(\mathsf{add}(d, s)) & = & s \\ \mathsf{one}(\epsilon) & = & \mathsf{false} \\ \mathsf{one}(\mathsf{add}(d, \epsilon)) & = & \mathsf{true} \\ \mathsf{one}(\mathsf{add}(d, \mathsf{add}(e, s))) & = & \mathsf{false} \end{array}$$

- a sort **Conf** of *confirmation messages*, with as domain the set

$$\{\mathsf{C\_OK}, \mathsf{C\_DONT\_KNOW}, \mathsf{C\_NOT\_OK}\}.$$

- a sort **Ind** of *indication messages*, with as domain the set

$$\{\mathsf{I\_OK}, \mathsf{I\_FIRST}, \mathsf{I\_INCOMPLETE}\}.$$

- a sort **Sstatus** of *status values of the sender*, with as domain the set

$$\{\mathsf{SF}, \mathsf{WA}, \mathsf{SC}, \mathsf{ET2}, \mathsf{WT2}\}.$$

- a sort **Rstatus** of *status values of the receiver*, with as interpretation the set

$$\{\mathsf{WF}, \mathsf{SI}, \mathsf{SA}, \mathsf{RTS}, \mathsf{NOT\_OK}\}.$$

We assume that all elements of the domains of **Conf**, **Ind**, **Sstatus** and **Rstatus** also occur in the language as constants symbols of the corresponding sorts.

3.2.2 Notation

In the presentation below, we will use the following conventions:

- We do not mention the precondition of transition types for input actions (since they are always equal to true).
- In the effect part of transition types we omit assignments of the form $x := x$.

- We write $\mathsf{if}\ c\ \mathsf{then}\ [z_1 := f_1, \ldots, z_k := f_k]$ as an abbreviation for

$$\begin{array}{rcl} z_1 & := & \mathsf{if}\ c\ \mathsf{then}\ f_1\ \mathsf{else}\ z_1 \\ & \vdots & \\ z_k & := & \mathsf{if}\ c\ \mathsf{then}\ f_k\ \mathsf{else}\ z_k \end{array}$$

- We never mention the partition of the local action types because in all I/O automata generators that we consider it is trivial in the sense that there is only a single block which contains all the action types.

3.2.3 Sender S

We will now present the I/O automaton S, which models the sender of the protocol. An important state variable of S is *status*, which gives the current status of the sender. This variable takes values in the domain **Sstatus**, which contains five elements:

- SF: Send a Frame at port F,
- WA: Wait for an Acknowledgement to arrive at port B,
- SC: Send a Confirmation message to the upper layer,
- ET2: Enable Timer 2, and
- WT2: Wait for Timeout of Timer 2.

We have modeled the arrival of a request (*REQ*) as an input action, since this event is clearly under control of the environment. However, once we have taken this decision the I/O automata model forces us to specify, for all possible states, what happens if an *REQ* action occurs. In our modeling, the sender discards an incoming request if it is busy handling the previous request, something which is recorded in the boolean state variable *busy*.

T3 is a time out event that can occur when S wants to send a frame into channel K but does not succeed because other agents (not specified here) are using the channel. After the occurrence of a *T3* action, S will send a confirmation message C_DONT_KNOW or C_NOT_OK.

When S sends a frame into channel K via an action F, it simultaneously starts a timer by setting boolean state variable *timer1_on* to true. This timer will timeout if an acknowledgement for the frame does not arrive in time. Since we cannot explicitly model real-time aspects in the I/O automata model, we deal with this timing behavior in a different way. Under the assumptions that (1) the transmission of a frame through channels K and L takes a bounded time, and (2) R will always acknowledge an incoming frame in a bounded time, and (3) the timer is set properly, a timeout will only occur if either the frame gets lost in channel K, or the acknowledgement for it gets lost in channel L. Thus one could say that the loss of a message in the channel "causes" a timeout event. In our specification we have made these causal links visible by introducing output actions *E1K* and *E1L* for channels K and L, respectively, which occur when a message gets lost, and corresponding input actions *E1K* and *E1L* of sender S, whose occurrence sets a boolean state variable *timer1_enabled*. By taking *timer1_enabled* to be part of the precondition of the timeout action *T1*, this gives us the desired causal links.

If something goes wrong during the handling of a request, and S sends a C_DONT_KNOW or C_NOT_OK confirmation message, then before dealing with a new request, S will wait long enough to make sure that the receiver R is prepared to receive new frames. Also here, since we cannot deal with real-time directly within our model, we describe the causal links that result from these real-time constraints. After sending a C_DONT_KNOW or C_NOT_OK confirmation message, the sender does an output action *E2*, which corresponds to starting a new timer (that is not specified here). Since it depends on the state of R when this timer will timeout, *E2* is made into an input action of R. At the appropriate moment R will generate the timeout action *T2* for the timer started by S, so that S can proceed and handle the next request.

We now give the code for I/O automaton S.

Input: *REQ*: **List**
B: **Bool** × **Bool** × **Bool**
E1K
E1L
T2

Output: *CONF*: **Conf**
F: **Bool** × **Bool** × **Bool** × **Data**
E2

Internal: *T1*
T3

State Variables:

status:	**Sstatus**
busy,first,toggle:	**Bool**
timer1_on:	**Bool**
timer1_enabled:	**Bool**
list:	**List**
rn:	**Nat**

Initialization: $status{=}\mathsf{SF} \wedge \neg busy \wedge first \wedge \neg timer1\_on \wedge \neg timer1\_enabled \wedge rn{=}0$

REQ(s)
Effect:
if $\neg busy \wedge s{\neq}\epsilon$ then $[list := s$
$busy := \mathsf{true}]$

$F(f, l, t, d)$
Precondition:
$status{=}\mathsf{SF} \wedge busy \wedge f{=}first \wedge l{=}\mathsf{one}(list) \wedge t{=}toggle \wedge d{=}\mathsf{hd}(list)$
Effect:
$status := \mathsf{WA}$
$timer1\_on := \mathsf{true}$
$rn := \mathsf{succ}(rn)$

T3
Precondition:
$status{=}\mathsf{SF} \wedge busy$
Effect:
$status := \mathsf{SC}$

E1K
Effect:
$timer1\_enabled := \mathsf{true}$

E1L
Effect:
$timer1\_enabled := \mathsf{true}$

$B(f, l, t)$
Effect:
$status :=$ if $\mathsf{one}(list)$ then SC else SF
$first := \mathsf{one}(list)$
$toggle := \neg toggle$
$timer1\_on := \mathsf{false}$
$list := \mathsf{tl}(list)$
if $\neg(\mathsf{one}(list))$ then $[rn := 0]$

T1
Precondition:
$timer1\_on \wedge timer1\_enabled$
Effect:
$status :=$ if $rn{\leq}\mathsf{max}$ then SF else SC
$timer1\_on := \mathsf{false}$
$timer1\_enabled := \mathsf{false}$

$CONF(c)$
  Precondition:
    $status$=SC
    $\wedge$ c=if $list$=ϵ then C_OK else (if one($list$) $\wedge$ $rn \neq 0$ then C_DONT_KNOW else C_NOT_OK)
  Effect:
    $status$:= if $list$=ϵ then SF else ET2
    $busy$:= false
    $list$:= ϵ
    rn := 0

$E2$
  Precondition:
    $status$=ET2
  Effect:
    $status$:= WT2
    $first$:= true
    $toggle$:= $\neg toggle$

$T2$
  Effect:
    $status$:= SF

3.2.4 Channel K

I/O automaton K gives a straightforward description of a faulty message buffer with capacity one. Messages that arrive when the buffer is full are discarded. However, in Lemma 3.2 we will show that such a situation never occurs during an actual run of the protocol.

Input: F: **Bool** $\times$ **Bool** $\times$ **Bool** $\times$ **Data**

Output: G: **Bool** $\times$ **Bool** $\times$ **Bool** $\times$ **Data**
$E1K$

State Variables: $full, first, last, toggle$: **Bool**
$datum$: **Data**

Initialization: $\neg full$

$F(f,l,t,d)$
  Effect:
    if $\neg full$ then [$full$:= true
      $first$:= f
      $last$:= l
      $toggle$:= t
      $datum$:= d]

$G(f,l,t,d)$
  Precondition:
    $full \wedge f{=}first \wedge l{=}last \wedge t{=}toggle \wedge d{=}datum$
  Effect:
    $full$:= false

$E1K$
  Precondition:
    $full$
  Effect:
    $full$:= false

3.2.5 Channel L

I/O automaton L is exactly the same as I/O automaton K, except that L handles frames that consist of 3 instead of 4 fields, and the actions have different names.

Input: A: **Bool** $\times$ **Bool** $\times$ **Bool**

Output: B: **Bool** $\times$ **Bool** $\times$ **Bool**
$E1L$

State Variables: *full,first,last,toggle*: **Bool** **Initialization**: $\neg$*full*

$A(f,l,t)$
Effect:
if $\neg$*full* then [*full* := true
first := f
last := l
toggle := t]

$B(f,l,t)$
Precondition:
full $\wedge$ f=*first* $\wedge$ l=*last* $\wedge$ t=*toggle*
Effect:
full := false

E1L
Precondition:
full
Effect:
full := false

3.2.6 Receiver R

I/O automaton R has a state variable *status*, whose value gives the current status of the receiver. The variable takes values in the domain **Rstatus**, which consists of five elements:

- WF: Wait for a Frame to arrive at port G,
- SI: Send an Indication message to the upper layer,
- SA: Send an Acknowledgement message at port A,
- RTS: Return the control bits of the received frame To the Sender via port A, and
- NOT_OK: send an indication message "NOT_OK" to the upper layer.

The subtle part in the definition of R is again the part concerned with timing. The receiver has a timer of its own, which it starts simultaneously with sending an acknowledgement message by setting the boolean state variable *timer2_on*. The timer will time out if no new frame arrives at port G for a sufficiently long time and it is clear that the sender has interrupted an attempt to transmit a list. When a timeout occurs, the receiver will set *ctoggle* to false (meaning that it will not reject the next frame on basis of its toggle bit), and it will generate an indication "NOT_OK" in the case some messages have not yet been received. Now if R has set the timer and S generates an *E2* event, then a transmission has been interrupted and a timeout event may occur. For convenience we identify, in this case, *E2* with the timeout event. If an *E2* event occurs and the receiver's timer has not been set, then this event should not be interpreted as a timeout, but just as a signal that a timeout event *T2* can be generated at the sender side.

We now present the code for R.

Input: G: **Bool** $\times$ **Bool** $\times$ **Bool** $\times$ **Data**
E2

Output: A: **Bool** $\times$ **Bool** $\times$ **Bool**
IND: **Data** $\times$ **Ind**
IND_NOT_OK
T2

State Variables:

status:	**Rstatus**
ffirst,flast,ftoggle:	**Bool**
fdatum:	**Data**
first,toggle,ctoggle:	**Bool**
timer2_on:	**Bool**
timer2_enabled:	**Bool**

Initialization: $status{=}\mathsf{WF} \wedge first \wedge \neg ctoggle \wedge \neg timer2\_on \wedge \neg timer2\_enabled$

$G(f, l, t, d)$
Effect:
if $status{=}\mathsf{WF}$ then $[status := \text{if } ctoggle \rightarrow t{=}toggle \text{ then SI else RTS}$
$ffirst := f$
$flast := l$
$ftoggle := t$
$fdatum := d$
if $ctoggle \rightarrow t{=}toggle$ then $[timer2\_on := \text{false}]\ ]$

$IND(d, i)$
Precondition:
$status{=}\mathsf{SI} \wedge d{=}fdatum$
$\wedge\ i{=}\text{if } flast \text{ then I\_OK else (if } ffirst \text{ then I\_FIRST else I\_INCOMPLETE)}$
Effect:
$status := \mathsf{SA}$
$first := flast$
$ctoggle := \text{true}$
$toggle := \neg ftoggle$

$A(f, l, t)$
Precondition:
$(status{=}\mathsf{SA} \vee status{=}\mathsf{RTS}) \wedge f{=}ffirst \wedge l{=}flast \wedge t{=}ftoggle$
Effect:
if $status{=}\mathsf{SA}$ then $[timer2\_on := \text{true}]$
$status := \mathsf{WF}$

IND_NOT_OK
Precondition:
$status{=}\mathsf{NOT\_OK}$
Effect:
$status := \mathsf{WF}$
$first := \text{true}$
$timer2\_on := \text{true}$

$E2$
Effect:
$timer2\_enabled := \text{true}$
if $timer2\_on$ then $[ctoggle := \text{false}$
if $\neg first$ then $[status := \mathsf{NOT\_OK}$
$timer2\_on := \text{false}]\ ]$

$T2$
Precondition:
$timer2\_enabled \wedge status{=}\mathsf{WF}$
Effect:
$timer2\_enabled := \text{false}$

3.2.7 The Bounded Retransmission Protocol (BRP)

The bounded retransmission protocol can now be defined as the parallel composition of automata S, K, L and R, with all communication between these components hidden:

$$BRP \triangleq \text{HIDE } I \text{ IN } (S \| K \| L \| R)$$

where

$$I \triangleq \{F(f,l,t,d), G(f,l,t,d), A(f,l,t), B(f,l,t), E1K, E1L, E2, T2 \mid f,l,t \text{ in domain } \mathbf{Bool}, d \text{ in domain } \mathbf{Data}\}$$

3.2.8 Correctness criterion P

We now specify the collection of allowed behaviors of the Bounded Retransmission Protocol in terms of an I/O automaton P. This automaton has the same input and output actions as BRP, but no internal actions. If a $REQ(s)$ action occurs in the initial state, then the regular behavior of P is to output the elements of s one by one, tagging the first datum with an indication I_FIRST, intermediate data with I_INCOMPLETE, and the last datum with I_OK. After sending the last datum the protocol will generate a confirmation message C_OK to indicate that the request has been carried out successfully, and return to its initial state. Requests that come in at a time when the previous request has not yet been processed, are always ignored. While a request is being processed, something may go wrong at any point and, instead of the C_OK message, a C_DONT_KNOW or a C_NOT_OK confirmation message may be sent. However, the C_DONT_KNOW message will only occur if at most one data element has not been delivered, whereas the C_NOT_OK will only occur if at least one data element has not been delivered. In case a C_NOT_OK or C_DONT_KNOW message is sent somewhere in the middle of the processing of a request, i.e., after the first but before the last data element has been delivered, the protocol will do an IND_NOT_OK action. After the IND_NOT_OK action the protocol returns to its initial state, except if it has just received a new request, which will then be processed.

Below we present the formal definition of I/O automaton P. In the next section we will prove that the BRP is indeed a correct implementation of P.

Input: REQ: **List**

Output: IND: **Data** $\times$ **Ind**
IND_NOT_OK
$CONF$: **Conf**

State Variables: $busy, first, error$: **Bool**
$list$: **List**

Initialization: $\neg busy \wedge first \wedge \neg error$

$REQ(s)$
Effect:
if $\neg busy \wedge s \neq \epsilon$ then $[busy := \mathsf{true}$
$list := s]$

$IND(d, i)$
Precondition:
$busy \wedge \neg error \wedge list \neq \epsilon \wedge d = \mathsf{hd}(list)$
$\wedge\ i = \mathsf{if\ one}(list)\ \mathsf{then\ I\_OK\ else\ (if}\ first\ \mathsf{then\ I\_FIRST\ else\ I\_INCOMPLETE)}$
Effect:
$first := \mathsf{one}(list)$
$list := \mathsf{tl}(list)$

$CONF(c)$
Precondition:
$busy \wedge \neg error$
$\wedge\ (c = \mathsf{C\_OK} \rightarrow list = \epsilon)$
$\wedge\ (c = \mathsf{C\_DONT\_KNOW} \rightarrow (list = \epsilon \vee \mathsf{one}(list)))$
$\wedge\ (c = \mathsf{C\_NOT\_OK} \rightarrow list \neq \epsilon)$
Effect:
$busy := \mathsf{false}$
$error := \neg first$
$list := \epsilon$

IND_NOT_OK
Precondition:
$error$
Effect:
$first := \mathsf{true}$
$error := \mathsf{false}$

3.3 Protocol Correctness Proof

3.3.1 Invariants

Before we can establish a weak refinement from BRP to P we must gain insight into what are the reachable states of BRP. To this end, we will now present 13 invariants of the protocol, i.e., properties that hold initially and that are preserved by transitions. Most of these invariants are proved by a routine induction on the length of the executions to the reachable states. In the manual proofs of the invariants, which together occupy about 16 pages of ASCII text, we used numbering of assertions, as advocated by Lamport [12], although, due to the fact that the proofs went rarely more than 4 levels deep, we found it easier to use explicit names, like 3.1.1.1, instead of the implicit ones, like $\langle 4\rangle 1$. As an illustration we have included the proof of the invariant $INVR$ (Lemma 3.6).

The first invariant, stated in Lemma 3.2, relates the control variables of the different components of the protocol. In order to distinguish between the state variables of the different components of BRP, we prefix each state variable by the name of the component it originates from.

Lemma 3.2 *The following property holds for all reachable states of BRP. INV1* $\triangleq$

$$
\begin{array}{ll}
S.status \in \{\mathsf{SF}, \mathsf{SC}, \mathsf{ET2}\} & \rightarrow\ R.status = \mathsf{WF} \\
\wedge & \\
S.timer1\_enabled & \rightarrow\ (S.status = \mathsf{WA} \wedge R.status = \mathsf{WF} \wedge \neg K.full \wedge \neg L.full) \\
\wedge & \\
K.full & \rightarrow\ (S.status = \mathsf{WA} \wedge R.status = \mathsf{WF} \wedge \neg L.full) \\
\wedge & \\
R.status \in \{\mathsf{SI}, \mathsf{SA}, \mathsf{RTS}\} & \rightarrow\ S.status = \mathsf{WA} \\
\wedge & \\
R.timer2\_enabled & \rightarrow\ (S.status = \mathsf{WT2} \wedge R.status \in \{\mathsf{WF}, \mathsf{NOT\_OK}\} \wedge \neg K.full \wedge \neg L.full) \\
\wedge & \\
L.full & \rightarrow\ (S.status = \mathsf{WA} \wedge R.status = \mathsf{WF} \wedge \neg K.full)
\end{array}
$$

Invariant $INV1$ already allows us to make several important observations on the behavior of the protocol. The invariant implies that sender S will never send a frame into channel K when the channel is busy delivering another frame. Similarly, receiver R will never send a frame into channel L when L already contains a frame. Thus the protocol does not need communication channels with a buffering capacity of more than one. Clause three and six together give that there will never be a message in both K and L at the same time. Thus, an implementation of the protocol may use a single bidirectional medium to implement both channels. If channel L delivers a frame to the sender S, then S is in fact waiting for this frame to arrive. Similarly, if channel K delivers a frame to receiver R, then the receiver is waiting for this frame.

It follows rather directly from invariant $INV1$ that in each reachable state of the protocol at most one of the four components enables a locally controlled action. This means that in a sense the protocol is fully sequential.

The second invariant $INVS$, stated in Lemma 3.3, gives some of relationships between the state variables of the sender that are valid in all reachable states. The proof is via a routine inductive argument and uses Lemma 3.2. Since the actions in which S does not participate trivially preserve the validity of $INVS$, one only has to establish that $INVS$ holds initially and is preserved by the actions of S.

Lemma 3.3 *The following property holds for all reachable states of BRP.* $INVS \triangleq$

$$\begin{array}{ll}
S.status{\in}\{\mathsf{WA},\mathsf{SC}\} & \rightarrow\ S.busy \\
\wedge & \\
S.status{=}\mathsf{WA} & \rightarrow\ (S.timer1_on \wedge S.rn{\neq}0) \\
\wedge & \\
S.list{=}\epsilon & \rightarrow\ ((S.status{\in}\{\mathsf{SF},\mathsf{ET2},\mathsf{WT2}\} \wedge \neg S.busy) \vee S.status{=}\mathsf{SC}) \\
\wedge & \\
S.status{\in}\{\mathsf{ET2},\mathsf{WT2}\} & \rightarrow\ S.rn{=}0 \\
\wedge & \\
S.rn{\neq}0 & \rightarrow\ S.busy \\
\wedge & \\
S.status{=}\mathsf{SC} \wedge S.list{=}\epsilon & \rightarrow\ S.first
\end{array}$$

Invariants $INVK$ and $INVRS$, stated in Lemma 3.4 and Lemma 3.5, deal with the flow of information from sender to receiver via channel K.

Lemma 3.4 *The following property holds for all reachable states of BRP.* $INVK \triangleq$

$$\begin{array}{ll}
K.full\ \rightarrow & (K.first{=}S.first \wedge K.last{=}\mathsf{one}(S.list) \\
& \wedge\ K.toggle{=}S.toggle \wedge K.datum{=}\mathsf{hd}(S.list))
\end{array}$$

Lemma 3.5 *The following property holds for all reachable states of BRP.* $INVRS \triangleq$

$$\begin{array}{ll}
R.status{\in}\{\mathsf{SI},\mathsf{SA},\mathsf{RTS}\}\ \rightarrow & (R.ffirst{=}S.first \wedge R.flast{=}\mathsf{one}(S.list) \\
& \wedge\ R.ftoggle{=}S.toggle \wedge R.fdatum{=}\mathsf{hd}(S.list))
\end{array}$$

The invariants $INVR$, $INVR'$ and $INVR''$ of Lemmas 3.6, 3.7 and 3.8, respectively, give certain relationships between the state variables of R that are valid in all reachable states.

Lemma 3.6 *The following property holds for all reachable states of BRP.* $INVR \triangleq$

$$\begin{array}{ll}
R.status{=}\mathsf{NOT_OK} & \rightarrow\ \neg R.ctoggle \\
\wedge & \\
R.status{=}\mathsf{SI} & \rightarrow\ (R.ctoggle \rightarrow R.ftoggle{=}R.toggle) \\
\wedge & \\
R.status{\in}\{\mathsf{RTS},\mathsf{SA}\} & \rightarrow\ R.ctoggle \wedge R.ftoggle{\neq}R.toggle
\end{array}$$

Proof: Let s' be a reachable state of BRP. By induction on the length n of the shortest execution of BRP that ends in s', we prove $s' \models INVR$. If $n = 0$, then s' is a start state. Hence $s' \models R.status{=}\mathsf{WF}$, which implies $s' \models INVR$.

For the induction step, suppose that s' is reachable via an execution with length $n{+}1$. Then there exists a state s that is reachable via an execution of length n and $s \xrightarrow{a} s'$, for some action a. By induction hypothesis, $s \models INVR$. We prove $s' \models INVR$ by a routine case distinction on a. In the proof we will use several times that, by Lemma 3.2, $s \models INV1$.

1. Assume a is an action in which R does not participate. Then $s' \models INVR$ trivially follows from $s \models INVR$ and the observation that a does not change any of the state variables mentioned in $INVR$.

2. Assume $a = G(f, l, t, d)$

2.1)	$s \models K.full$	(by 2 and precondition G)
2.2)	$s \models R.status{=}\mathsf{WF}$	(by 2.1 and $INV1$)
2.3)	Assume $s \models R.ctoggle \rightarrow t{=}R.toggle$	
2.3.1)	$s' \models R.ctoggle \rightarrow t{=}R.toggle$	(by 2 and 2.3 since G does not change $R.ctoggle$ and $R.toggle$)
2.3.2)	$s' \models R.status{=}\mathsf{SI} \wedge R.ftoggle{=}t$	(by $2, 2.2, 2.3$ and effect G)
2.3.3)	$s' \models INVR$	(by 2.3.1 and 2.3.2)
2.4)	Assume $s \models \neg(R.ctoggle \rightarrow t{=}R.toggle)$	
2.4.1)	$s' \models \neg(R.ctoggle \rightarrow t{=}R.toggle)$	(by 2 and 2.4 since G does not change $R.ctoggle$ and $R.toggle$)
2.4.2)	$s' \models R.status{=}\mathsf{RTS} \wedge R.ftoggle{=}t$	(by $2, 2.2, 2.4$ and effect G)
2.4.3)	$s' \models INVR$	(by 2.4.1 and 2.4.2)
2.5)	$s' \models INVR$	(by 2.3 and 2.4)

3. Assume $a = IND(d, i)$

3.1)	$s' \models R.status{=}\mathsf{SA} \wedge R.ctoggle \wedge R.ftoggle{\neq}R.toggle$	(by 3 and effect IND)
3.2)	$s' \models INVR$	(by 3.1)

4. Assume $a = A(f, l, t)$

4.1)	$s' \models R.status{=}\mathsf{WF}$	(by 4 and effect A)
4.2)	$s' \models INVR$	(by 4.1)

5. Assume $a = E2$

5.1)	$s \models S.status{=}\mathsf{ET2}$	(by 5 and precondition $E2$)
5.2)	$s \models R.status{=}\mathsf{WF}$	(by 5.1 and $INV1$)
5.3)	Assume $s \models R.timer2\_on \wedge \neg R.first$	
5.3.1)	$s' \models R.status{=}\mathsf{NOT\_OK} \wedge \neg R.ctoggle$	(by $5, 5.3$ and effect $E2$)
5.3.2)	$s' \models INVR$	(by 5.3.1)
5.4)	Assume $s \models \neg(R.timer2\_on \wedge \neg R.first)$	
5.4.1)	$s' \models R.status{=}\mathsf{WF}$	(by $5, 5.2, 5.4$ and effect $E2$)
5.4.2)	$s' \models INVR$	(by 5.4.1)
5.5)	$s' \models INVR$	(by 5.3 and 5.4)

6. Assume $a = T2$

6.1)	$s \models R.status{=}\mathsf{WF}$	(by 6 and precondition $T2$)
6.2)	$s' \models R.status{=}\mathsf{WF}$	(by $6, 6.1$ and effect $T2$)
6.3)	$s' \models INVR$	(by 6.2)

7. Assume $a = IND\_NOT\_OK$

7.1)	$s' \models R.status{=}\mathsf{WF}$	(by 7 and effect IND_NOT_OK)
7.2)	$s' \models INVR$	(by 7.1)

8. $s' \models INVR$ (by 1-7) ∎

Lemma 3.7 *The following property holds for all reachable states of BRP.* $INVR' \triangleq$

$$\begin{array}{lcl} R.status{=}\mathsf{WF} & \rightarrow & (R.ctoggle \rightarrow R.timer2\_on) \\ \wedge & & \\ R.timer2\_on & \rightarrow & R.status{\in}\{\mathsf{WF},\mathsf{RTS}\} \\ \wedge & & \\ R.status{=}\mathsf{RTS} & \rightarrow & R.timer2\_on \end{array}$$

Lemma 3.8 *The following property holds for all reachable states of BRP.* $INVR'' \triangleq$

$$R.status{\in}\{\mathsf{WF},\mathsf{RTS}\} \;\rightarrow\; (R.first \vee R.timer2\_on)$$

The next invariant implies that when an acknowledgement message arrives at the sender, the three bits of this acknowledgement are determined by the state of the sender, and hence provide no information. The only information conveyed by an acknowledgement is the fact of its arrival itself.

Lemma 3.9 *The following property holds for all reachable states of BRP.* $INVL \triangleq$

$$\begin{array}{lcl} L.full & \rightarrow & (L.first{=}R.ffirst{=}S.first \wedge L.last{=}R.flast{=}\mathsf{one}(S.list) \\ & & \wedge\; R.ctoggle \wedge L.toggle{=}\neg R.toggle{=}S.toggle) \end{array}$$

Lemma 3.10 *The following property holds for all reachable states of BRP.* $INVS' \triangleq$

$$\begin{array}{lcl} (S.status{=}\mathsf{SC} \wedge S.list{=}\epsilon) & \rightarrow & (R.ctoggle \wedge S.toggle{=}R.toggle) \\ \wedge & & \\ (S.status{\in}\{\mathsf{SF},\mathsf{SC}\} \wedge S.rn{=}0) & \rightarrow & (R.ctoggle \rightarrow S.toggle{=}R.toggle) \\ \wedge & & \\ S.status{=}\mathsf{WT2} & \rightarrow & \neg R.ctoggle \end{array}$$

Lemma 3.11 *The following property holds for all reachable states of BRP.* $INVS'' \triangleq$

$$S.list{=}\epsilon \;\rightarrow\; (S.status{=}\mathsf{ET2} \vee (R.ctoggle \rightarrow S.toggle{=}R.toggle))$$

Proof: By combination of $INVS$ and $INVS'$. ■

Lemma 3.12 *The following property holds for all reachable states of BRP.* $INVFIRST \triangleq$

$$\begin{array}{lcl} R.status{=}\mathsf{NOT\_OK} & \rightarrow & S.first \\ \wedge & & \\ (S.rn{\neq}0 \wedge R.ctoggle \wedge S.toggle{\neq}R.toggle) & \rightarrow & (R.first{=}\mathsf{one}(S.list) \wedge S.first{=}R.ffirst) \\ \wedge & & \\ ((R.ctoggle \rightarrow S.toggle{=}R.toggle) \wedge R.status{\neq}\mathsf{NOT\_OK}) & \rightarrow & S.first{=}R.first \end{array}$$

Lemma 3.13 *The following property holds for all reachable states of BRP.* $INVRFIRST \triangleq$

$$\begin{array}{lcl} (S.status{=}\mathsf{SC} \wedge S.list{=}\epsilon) & \rightarrow & R.first \\ \wedge & & \\ R.status{=}\mathsf{SI} & \rightarrow & R.first{=}R.ffirst \end{array}$$

Proof: By combination of $INV1, INVS, INVS', INVR, INVRS$ and $INVFIRST$. ■

The next and final invariant is not used in the proof of the refinement, but interesting because it implies that, when a frame arrives at the receiver, the first field of this frame is determined by the state of the receiver and the other fields of the frame. Hence the first bit of the frame conveys no information and is redundant.

Lemma 3.14 *The following property holds for all reachable states of BRP.* $INVK' \triangleq$

$$K.full \;\rightarrow\; K.first = \mathsf{if}\ R.ctoggle \rightarrow K.toggle = R.toggle\ \mathsf{then}\ R.first\ \mathsf{else}\ R.ffirst$$

Proof: By combination of $INV1, INVK, INVS$ and $INVFIRST$. ■

3.3.2 The refinement relation

We have now prepared the ground for one of the main results of this paper, the refinement relation from BRP to P.

Theorem 3.15 *The function determined by the following formula is a weak refinement from BRP to P.* $REF \triangleq$

$$\begin{array}{lcl} P.busy & = & S.busy \\ \wedge & & \\ P.first & = & R.first \\ \wedge & & \\ P.error & = & R.status{=}\mathsf{NOT\_OK} \vee (S.status{=}\mathsf{ET2} \wedge R.timer2\_on \wedge \neg R.first) \\ \wedge & & \\ P.list & = & \mathsf{if}\ S.status{=}\mathsf{ET2} \vee (R.ctoggle \rightarrow S.toggle{=}R.toggle)\ \mathsf{then}\ S.list\ \mathsf{else}\ \mathsf{tl}(S.list) \end{array}$$

Proof: Not included in this paper. Given the invariants established above, the proof is a routine exercise, which however still takes almost 5 pages densely filled with ASCII. ■

3.3.3 Absence of deadlock

In this subsection we will establish that the BRP does not have *deadlocks*, i.e., states in which the system is quiescent even though it should not be so according to the specification. Because in I/O automata input actions are always enabled, they will typically not have deadlock states in the sense of states without any outgoing transitions. Instead we define an I/O automaton A to be *deadlock free with respect to* an I/O automaton B if $qtraces(A) \subseteq qtraces(B)$. This means that whenever it is possible to reach a quiescent state of A via some trace, we can also reach a quiescent state of B with the same trace.

In order to prove that BRP is deadlock free with respect to P, we need one additional invariant.

Lemma 3.16 *The following property holds for all reachable states of BRP.* $INVD \triangleq$

$$\begin{array}{lcl} S.status{=}\mathsf{WT2} & \rightarrow & R.timer2\_enabled \\ \wedge & & \\ R.status{=}\mathsf{NOT\_OK} & \rightarrow & S.status{=}\mathsf{WT2} \\ \wedge & & \\ (S.status{=}\mathsf{WA} \wedge R.status{=}\mathsf{WF}) & \rightarrow & (K.full \vee L.full \vee S.timer1\_enabled) \end{array}$$

Theorem 3.17 *BRP is deadlock free with respect to P.*

Proof: It is sufficient to prove that for each reachable and quiescent state s of BRP, $REF(s)$ is a quiescent state of P. Since REF is a weak refinement from BRP to P this will imply that BRP is deadlock free with respect to P.

1)	$s \models R.status \neq \mathsf{SI}$	(since $s \models \neg enabled(IND)$)
2)	$s \models R.status \neq \mathsf{SA} \wedge R.status \neq \mathsf{RTS}$	(since $s \models \neg enabled(A)$)
3)	$s \models R.status \neq \mathsf{NOT\_OK}$	(since $s \models \neg enabled(IND\_NOT\_OK)$)
4)	$s \models R.status = \mathsf{WF}$	(by $1, 2$ and 3)
5)	$s \models S.status \neq \mathsf{SC}$	(since $s \models \neg enabled(CONF)$)
6)	$s \models S.status \neq \mathsf{ET2})$	(since $s \models \neg enabled(E2)$)
7)	$s \models \neg R.timer2\_enabled$	(by 4 and $s \models \neg enabled(T2)$)
8)	$s \models S.status \neq \mathsf{WT2}$	(by 7 and $s \models INVD$)
9)	$s \models Sstatus = \mathsf{WA} \rightarrow S.timer1\_on$	(by $INVS$)
10)	$s \models \neg(S.timer1\_on \wedge S.timer1\_enabled)$	(by $\neg enabled(T1)$
11)	$s \models \neg K.full$	(since $s \models \neg enabled(G)$)
12)	$s \models \neg L.full$	(since $s \models \neg enabled(B)$)
13)	$s \models S.status \neq \mathsf{WA}$	(by $4, 9, 10, 11, 12$ and $s \models INVD$)
14)	$s \models S.status = \mathsf{SF}$	(by $5, 6, 8$ and 13)
15)	$s \models \neg(S.status = \mathsf{SF} \wedge S.busy)$	(since $s \models \neg enabled(T3)$)
16)	$s \models \neg S.busy$	(by 14 and 15)
17)	$REF(s) \models \neg P.busy \wedge \neg P.error$	(by $4, 14, 16$ and definition REF)
18)	$REF(s) \models quiescent(P)$	(by 17 and inspection of local transition types P) ■

3.3.4 Inclusion of fair traces

We now come to the main result of this section, which says that the Bounded Retransmission Protocol is a correct implementation of the specification automaton P.

Theorem 3.18 *BRP implements P.*

Proof: (Sketch) By Theorem 3.15, we know that REF maps each execution of BRP to an execution of P. We will show that REF moreover maps each fair execution of BRP to a fair execution of P. This then immediately implies the theorem.

By the proof Theorem 3.17, REF maps each finite fair execution of BRP to a finite fair execution of P. Thus it is enough to prove that REF maps each infinite fair execution of REF to an infinite fair execution of P. ■

4 Proof-Checking

4.1 Coq Proof Development System

Coq is a proof assistant for higher-order logic. It is based on the *Calculus of Inductive Constructions* [18], which is a polymorphic type theory allowing dependent types and inductive types. It is based on [6]. Constructing a proof in Coq is an interactive process. The user specifies which deduction rule should be applied and Coq does all the calculations and bookkeeping.

4.1.1 The Tactics Theorem Prover

The system makes use of the *Curry-Howard isomorphism*, which states that λ-terms can be used to encode natural deduction proofs. For instance the well-known S-combinator

$$\lambda x : A \to (B \to C).\ \lambda y : A \to B.\ \lambda z : A.\ xz(yz)$$

encodes under the Curry-Howard isomorphism the following natural deduction proof. (Cancelled hypotheses are placed between square brackets.)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\,A \to (B \to C)\,]^3 \quad [\,A\,]^1}{B \to C} \qquad \dfrac{[\,A \to B\,]^2 \quad [\,A\,]^1}{B}}{C}}{A \to C}\,1}{(A \to B) \to (A \to C)}\,2}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,3$$

In order to give the reader a flavor of a proof session in `Coq` we give the list of commands needed to construct the proofterm above. At the right we expose how the proofterm is built step by step. Note that the proofterm is constructed in a top-down fashion. The terms `Hyp1`, …, `Hyp5` are meta-variables over proofterms for subgoals that are generated during the proof session. They are instantiated during goal refinement. (We omit the types in the proofterm in order to save space.)

`Goal (A->(B->C))->((A->B)->(A->C)).`	proofterm: `Hyp1`
`Intros x y z.`	proofterm: λ`xyz.Hyp2`
`Apply x.`	proofterm: λ`xyz.x Hyp3 Hyp4`
`Assumption.`	proofterm: λ`xyz.x z Hyp4`
`Apply y.`	proofterm: λ`xyz.x z (y Hyp5)`
`Assumption.`	proofterm: λ`xyz.x z (y z)`

Commands (*tactics*) can be composed to so called *tacticals*. The tactical `tac0 ; tac1` first applies `tac0` on the current goal and then applies `tac1` on all the subgoals generated by `tac0`. More generally, the tactical `tac0 ; [ tac1 | ··· | tacN ]` first applies `tac0` and then applies `taci` on the i-th subgoal generated by `tac0` ($i = 1, \ldots$ `N`). (When `tac0` does not generate `N` subgoals, this tactical fails.) The following tactical generates the same S-combinator.

```
Intros x y z ; Apply x ; [ Assumption | Apply y ; Assumption ].
```

Details about the use of `Coq` can be found in the `Coq` manual [7].

One of the most important features of `Coq` is the so called *program abstraction*. From a proof of $\forall x : A.\ \exists y : B.\ P(x, y)$ one can extract a function (program) $f : A \longrightarrow B$ such that $\forall x : A.\ P(x, f(x))$. We do not need this facility for our purposes.

4.1.2 Inductive Types

In our encodings we extensively use the inductive types. For details about this phenomenon we refer to [18]. In this paper we restrict ourselves to some examples. When we define

```
nat  :  Set
O    :  nat
S    :  nat->nat
```

then $\underbrace{\texttt{S (S (}\cdots\texttt{(S (S}}_{n}\texttt{ O))}\cdots\texttt{))}$ is of type `nat` for all $n \in \mathcal{N}$ but there might still be other terms of type `nat`. In Coq we have the alternative possibility

```
nat  :=  Ind(X:Set){ X | X->X }
O    :=  Constr(1,nat)
S    :=  Constr(2,nat).
```

This must be read as '`nat` is the smallest set `X` closed under two constructors, one of type `X` and one of type `X->X`'. When we choose for the second option then `nat` contains no other terms then those constructed from `O` and `S`. In other words: for an arbitrary term `P:nat->`$*$ and an arbitrary `x:nat` we are able to construct a term of type `(P x)` from terms $\wp_o$`:(P O)` and $\wp_s$`:`Π`y:nat.(P y)->(P (S y))`. This term is written as `(<P>Match x with` $\wp_o$ $\wp_s$`)` in the system. The reduction behavior of this term is determined by the construction of `x` from `O` and `S`.

$$
\begin{array}{lll}
\texttt{<P>Match O with } \wp_o\ \wp_s & \twoheadrightarrow & \wp_o \\
\texttt{<P>Match (S y) with } \wp_o\ \wp_s & \twoheadrightarrow & \wp_s\ \texttt{y (<P>Match y with } \wp_o\ \wp_s\texttt{)}
\end{array}
$$

Note that these reductions are well typed, i.e. reduction of a term does not change its type. When $* \equiv$ `Prop` (which is a predefined notion of Coq, representing the type of all propositions) then `P` is a predicate over `nat` and $\wp_o$ and $\wp_s$ are just the usual proofs for the zero-case and the successor-case. When $* \equiv$ `Set` (another predefined notion, representing the type of all sets) and `P` is a constant function on `nat`, say `P` $\equiv \lambda$`n:nat.A` for some `A:Set`, then $\wp_o$`:A` and $\wp_s$`:nat->A->A` and λ`x:nat.(<P>Match x with` $\wp_o$ $\wp_s$`)` represents the function from `nat` to `A` that is defined by primitive recursion from $\wp_o$ and $\wp_s$. In other words: λ`a:A.`λ`g:nat->A->A.`λ`x:nat.(<P>Match x with a g)` is a recursor. With this mechanism one can define any primitive recursive function (and even more because one can use higher order recursion).

We conclude this subsection with the illustration of how one can use inductive types to encode the logical symbols $\wedge$ and $\vee$. Define

```
and        :=  λA,B:Prop.Ind(X:Prop){ A->B->X }
or         :=  λA,B:Prop.Ind(X:Prop){ A->X | B->X }
conj       :=  λA,B:Prop.Constr(1,(and A B))
or_introl  :=  λA,B:Prop.Constr(1,(or A B))
or_intror  :=  λA,B:Prop.Constr(2,(or A B))
```

then `(and A B)` contains no other terms (proofs) then those constructed from `(conj A B)` and `(or A B)` contains no other terms then those constructed from `(or_introl A B)` and (or_intror A B). This exactly reflects the intuitionistic meanings of $\wedge$ and $\vee$. The only way to prove $A \wedge B$ is proving both A and B, and the only way to prove $A \vee B$ is proving A or B.

Remark: In this paper we write λ's and Π's in Coq input and Coq output in order to improve the readability. In the system these symbols are replaced by square brackets and parenthesis, so [x:A]b instead of λx:A.b and (x:A)B instead of Πx:A.B.

The hand-written proof is written in many sorted predicate logic. For each sort there is an equality relation. We use the standard encoding

```
eq  :  ΠA:Set.A->A->Prop
```

to represent these equality. These encoding can be found in [7]. Note that eq is a polymorphic equality. Furthermore we used the standard encodings and and or, briefly explained in the previous subsection. The types Prop and Set, also mentioned in the previous subsection, are predefined notions (constants) of Coq, comparable with '$*$' in systems of Barendregt's λ-cube [2]. The logical implication and the functional implication are both identified with the arrow of type theory. (As a consequence our proof is intuitionistically valid.)

There are at least two ways to encode the functional behavior of a function $F : A \longrightarrow B$. For instance the sum $+ : \mathbf{Nat} \longrightarrow \mathbf{Nat} \longrightarrow \mathbf{Nat}$ can be defined by

```
sum   :  nat->nat->nat
sum1  :  Πx:nat.<nat>(sum O x)=x
sum2  :  Πx,y:nat.<nat>(sum (S y) x)=(S (sum y x))
```

where <A>a=b is syntactic sugaring for (eq A a b). In this case sum is just a variable without any computational power. Computing the value of (sum n m) can be done by the Coq command 'Rewrite sum1.' or 'Rewrite sum2.' depending on the value of n. The alternative is to define sum as an abbreviation.

```
sum := λx,y:nat.<λz:nat.nat>Match x with y λz:nat.S.        (1)
```

The advantage of the second approach is that one does not have to give any command for computing the value of (sum n m). The computation of (sum n m) is just normalization of (sum n m), which is executed automatically by the system. Note that Coq cannot reduce (sum n O)) to n when n is a variable. In such a case one can do a case analysis on n. We try to use the second approach as much as possible.

In Coq it is allowed to omit the λ-abstraction in <P>Match ... when P is a constant predicate, so <nat>Match ... instead of <λz:nat.nat>Match ... in the definition of sum. In the sequel we will omit such λ-abstractions.

The main result in the hand-written proof is the claim that there exists a *weak refinement* from automaton BRP to automaton P. We modified our encodings several times in order to get a better formulation in Coq of this weak refinement property. Furthermore we would like to be able to represent functions like REF:(states of BRP) $\longrightarrow$ (states of P) by a λ-term like in (1).

4.1.3 The Datatypes

The specification of the Bounded Retransmission Protocol makes use of several datatypes. We represent these types by inhabitants of `Set`.

- the sort **Bool** is represented by the inductive type

  ```
  bool   :=  Ind(X:Set){ X | X }
  true   :=  Constr(1,bool)
  false  :=  Constr(2,bool)
  ```

  The standard functions $\neg$, $\wedge$, $\vee$, $\rightarrow$, $=$ and $\neq$ on booleans can all be represented by λ-terms. For instance $\wedge$ can be represented by

  ```
  andb  :=  x,y:bool.<bool>Match x with
              <bool>Match y with true false
              false
  ```

- The sort **Data** is represented by the variable `data:Set`. Furthermore we define a variable `Undefined:data` which represents the element $\perp \in$ **Data**.

- The sort **List** is defined as the inductive type with constructors `NIL` and `CONS`, representing ϵ and add respectively. In formula:

  ```
  LIST  :=  Ind(X:Set){ X | data->X->X }
  NIL   :=  Constr(1,LIST)
  CONS  :=  Constr(2,LIST)
  ```

  Functions like hd, tl, ϵ and one can all be represented by λ-terms. For instance

  ```
  tl   :=  λL:LIST.<LIST>Match L with
             NIL
             λd:data.λy,z:LIST.y
  one  :=  λL:LIST.<bool>Match L with
             false
             λd:data.λy:LIST.λb:bool.<bool>Match y with
               true
               λd:data.λy:LIST.λb:bool.false
  ```

  The equalities on page 182 are satisfied. All the right-hand-sides are just the normal forms of the left-hand-sides.

- The finite sets **Ind**, **Sstatus**, **Rstatus** and **Conf** are encoded as inductive types in the same style as the booleans.

4.1.4 The Actions

We define finite sets `act_BRP` and `act_P` representing the sets of actions. We cannot use the same name for actions of different automata. Hence we add a prime by those action of automaton P that already occurred in automaton BRP. Constructors of `act_BRP` are `REQ:LIST->act_BRP`, `F:bool->bool->bool->data->act_BRP`, etc. Elements of `act_P` are `REQ':LIST->act_P`, etc.

We add an extra element `tau` to the inductive set `act_P`. Next we define a term `ev` (evaluate) which maps actions of BRP to the corresponding actions of P. (`REQ` $\overset{\texttt{ev}}{\longmapsto}$ `REQ'`, etc.) Internal actions of BRP are mapped to `tau`.

```
ev := λa:act_BRP.<act_P>Match a with  REQ'
                                      λB1,B2,B3:bool.λd1:data.tau
                                      tau
                                      λB1,B2,B3:bool.tau
                                      tau
                                      tau
                                      tau
                                      CONF'
                                      tau
                                      tau
                                      λB1,B2,B3:bool.λd1:data.tau
                                      λB1,B2,B3:bool.tau
                                      IND'
                                      INDn'.
```

Note that we simply posit which actions are internal in automaton BRP. One could think of encoding the complete theory about input- and output actions. Given the status of the actions in the components, the status of the actions in the product automaton could then be computed. However, a Coq formalization of this part of automaton theory is arduous and not cost-effective.

4.1.5 The State Spaces

The following step is the definition of types `states_BRP` and `states_P` representing the state spaces of the two automata BRP and P. The state space of BRP is encoded as a cartesian product of cartesian products. `states_S` := `status_S` × `bool` × ⋯ × `bool` × `LIST` × `nat`, where `status_S` = {`SF`, `WA`, `SC`, `ET2`, `WT2`}. Analogously we define `states_K`, `states_L` and `states_R`. Now `states_BRP` := `states_S` × `states_K` × `states_L` × `states_R`. Finally `states_P` := `LIST` × `bool` × `bool` × `bool`.

We use the standard inductive type `prod` [7] with constructor `pair` for the encoding of the cartesian product. When A and B are sets and $(a, b) \in A \times B$ then (a, b) is represented in Coq by `(pair A B a b)`. Hence $(a, b, c) \in A \times B \times C$ is represented by `(pair A (prod B C) a (pair B C b c))` which introduces unattractive syntax. For instance, an element of `states_BRP` would cover the whole page. However, we can handle big product terms by defining abbreviations for them. For instance, we define a function $F : A \longrightarrow B \longrightarrow C \longrightarrow (A \times B \times C)$ such that $F(a)(b)$ maps c to (a, b, c) by a λ-term in the style of (1).

```
F  :=  λx:A,y:B,z:C.(pair A (prod B C) x (pair B C y z))
```

Then (a, b, c) can be represented by `(F a b c)`. This way we define functions `st_S`, `st_K`, `st_L`, `st_R`, `st_BRP` and `st_P` mapping the components of the state spaces to the corresponding elements in the cartesian products. Similarly we define projection functions `p_t_S:states_S->bool`, `p_R:states_BRP->states_R`, etc. (`p_t_S` stands for *projection_toggle_Sender* and `p_R` for *projection_Receiver*). For instance

```
(p_t_S (st_S B1 B2 B3 B4 B5 L rn))   ↠  B3
(p_R (st_BRP S K L R))               ↠  R
```

4.1.6 The Weak Refinement Map

The refinement mapping REF can now be represented by the λ-term below. We replace the lambda-terms `orb`, `andb`, `negb`, `eqb`, `eqst_S` and `eqst_R` by $\vee$, $\wedge$, $\neg$, $=$, $=$ and $=$ respectively. Furthermore we used infix-notation.

```
ref  :=  λx:states_BRP.(st_P
           (<LIST>Match
             (p_st_S (p_S x)) = ET2 ∨
             ¬ (p_ct_R (p_R x)) ∨
             (p_t_S (p_S x)) = (p_t_R (p_R x))
           with
             (p_l_S (p_S x))
             (tl (p_l_S (p_S x))))

           (p_b_S (p_S x))

           (p_f_R (p_R x))

           (p_st_R (p_R x)) = NOK ∨
           (((p_st_S (p_S x)) = ET2 ∧
           (p_on_R (p_R x)) ∧
           ¬ (p_f_R (p_R x)))).
```

4.1.7 The Step Relation

The next step is the definition of types `step:act_BRP->states_BRP->states_BRP->Prop` and `step':act_P->states_P->states_P->Prop` representing the notion of transition step. The intended meaning of (`step a s1 s2`) is $s_1 \xrightarrow{a} s_2$. We use an inductive type again.

```
step  :=  Ind(X:act_BRP->states_BRP->states_BRP->Prop){
            Πsigma:LIST.ΠS1:status_S.ΠB1,B2,B3,B4:bool.
            ΠL:LIST.Πrn:nat.
            ΠsK:states_K.ΠsL:states_L.ΠsR:states_R.
              (<bool>false=(empty sigma)) ->
                (X (REQ sigma)
                  (st_BRP (st_S S1 false B1 B2 B3 B4 L rn) sK sL sR)
                  (st_BRP (st_S S1 true B1 B2 B3 B4 sigma rn) sK sL sR))
        |
        ⋮
        |   ··· }
```

This enables us to do a case analysis on `H` when we have a proof `H:(step a s1 s2)` in our context. Assume that we want to prove ϕ(`s1,s2`) for some `s1,s2:states_BRP` and assume that we have a proof `H:(step a s1 s2)` in our context. The first constructor of `step` leads to the following subgoal:

```
Πsigma:LIST.ΠS1:status_S.ΠB1,B2,B3,B4:bool.
ΠL:LIST.Πrn:nat.
  <bool>false=(empty sigma) ->
  φ((st_BRP (st_S S1 false B1 B2 B3 B4 L rn) sK sL sR),
    (st_BRP (st_S S1 true B1 B2 B3 B4 sigma rn) sK sL sR))
```

This is the result of matching `s1` and `s2` with the terms of type `states_BRP` on which `X` is applied in the first–constructor–case of `step` (describing the behavior of the *REQ* action).

In our approach we encode directly how the actions affect the product automaton BRP. This way we avoid the problem of encoding how the composition of the automaton BRP out of its components S, K, L and R is organized. The fact that local actions that have the same name are synchronized in the product automaton is difficult to express.

In Coq, actions with *if-then-else* constructions are split in more than one case. This way, the 15 actions in Section 3.2 correspond to 25 cases in Coq. For instance the action B is split into B_1 with precondition one(*list*)=true and B_2 with precondition one(*list*)=false. (Both B_1 and B_2 also have all the preconditions mentioned in automaton L).

4.1.8 Reachability

Reachability is encoded as an inductive type, having two constructors. The first constructor encodes the reachability of the initial state. The second constructor encodes the preservation of `reach` under `step`.

```
reach  :=  Ind(X:states_BRP->Prop){
             Πs:states_BRP.(start s) -> (X s)
       |     Πa:act_BRP.Πs1,s2:states_BRP.
               (step a s1 s2) -> (X s1) -> (X s2) }
```

where `start` is the predicate on `states_BRP` that holds only for the initial state, also defined inductively:

```
start  :=  Ind(X:states_BRP->Prop){
             ΠB1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11:bool.
             ΠL:LIST.λd1,d2:data.
               (X (st_BRP
                 (st_S SF false true B1 false false L O)
                 (st_K B2 B3 B4 false d1)
                 (st_L B5 B6 B7 false)
                 (st_R WF B9 B10 B11 d2 true B8 false false false))) }.
```

Assume that we want to prove ϕ(s) for some `s:states_BRP` and assume that we have a proof `R:(reach s)`. Eliminating the inductive type `reach` returns two subgoals:

```
(i)   :  Πs1:states_BRP.(start s1) -> φ(s1)
(ii)  :  Πa:act_BRP.Πs1:states_BRP. (step a s1 s) -> φ(s1,s2) -> (reach s1) -> φ(s).
```

The second goal can be proved by assuming an `a:act_BRP`, `s1`, `s2` and a proof `H:(step a s1 s)` and then prove ϕ(`s1`) `->` ϕ(`s2`) by a case analysis on `H`.

We can define a type `reach'` encoding the reachability in the automaton P by using `start'` and `step'` but we don't need this notion.

4.1.9 The Weak Refinement Property

Assume that we have a transition $s_1 \xrightarrow{a} s_2$ for some external BRP-action a, then we must have a transition $\mathrm{REF}(s_1) \xrightarrow{a} \mathrm{REF}(s_2)$ in automaton P. We are able to express this as

```
(step a s1 s2) -> (step' (ev a) (ref s1) (ref s2))          (2)
```

When a is an internal action then (2) evaluates to

```
(step a s1 s2) -> (step' tau (ref s1) (ref s2))
```

which we cannot prove for there are no constructors of the form `step' tau` ... in the definition of `step'`. When we add a constructor of type `Πs:states_P.(step' tau s s)` then we can prove `(step a s1 s2) -> (step' tau (ref s1) (ref s2))` iff we can prove `<states_P>(ref s1)=(ref s2)`. This is exactly what we required so (2) also encodes the weak refinement property when a is internal.

Of course (2) does not have to hold for states that cannot be reached. Hence we can add an extra precondition. Furthermore we abstract from the states and the action:

```
Πa:act_BRP.Πs1,s2:states_BRP.(reach s1) ->
(step a s1 s2) -> (step' (ev a) (ref s1) (ref s2))          (3)
```

A weak refinement mapping also has to map initial states to initial states. This is encoded as

```
Πs:states_BRP.(start s) -> (start' (ref s))          (4)
```

4.1.10 The Invariants

For proving (3) we have to use the invariants. These invariants are proven valid in the reachable states only. Their formulation is rather straightforward. Below we give the encoded version of *INVR* (Lemma 3.6). Note that the expression has precondition `(reach x)`. Furthermore $x \in \{y_1, \ldots y_n\}$ is encoded as $x = y_1 \vee \cdots \vee x = y_n$. (Again, we give a 'pretty-printed' version)

```
invr   :=  Πx:states_BRP.(reach x) ->
           ((p_st_R (p_R x)) = NOK ->
             (p_ct_R (p_R x)) = false)
       ∧
           ((p_st_R (p_R x)) = SI ->
             ((p_ct_R (p_R x)) = true ->
               (p_ft_R (p_R x)) = (p_t_R (p_R x))))
       ∧
           (((p_st_R (p_R x)) = RTS ∨
               (p_st_R (p_R x)) = SA) ->
             ((p_ct_R (p_R x)) = true ∧
               (p_ft_R (p_R x)) = ¬ (p_t_R (p_R x)))).
```

Machine-checking the invariant proofs can be done with a less complex definition of reachability. Instead of having separate definitions `step` and `reachability` one can construct one single definition `reach_alt` (*reach_alternative*) which has the following shape:

```
reach_alt  :=  Ind(X:states_BRP->Prop){
                 ΠB1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11:bool.
                 ΠL:LIST.λd1,d2:data.
                     (X (st_BRP
                       (st_S SF false true B1 false false L 0)
                       (st_K B2 B3 B4 false d1)
                       (st_L B5 B6 B7 false)
                       (st_R WF B9 B10 B11 d2 true B8 false false false)))
                 Πsigma:LIST.ΠS1:status_S.ΠB1,B2,B3,B4:bool.
                 ΠL:LIST.Πrn:nat.
            |    ΠsK:states_K.ΠsL:states_L.ΠsR:states_R.
                   (<bool>false=(empty sigma)) ->
                     (X (st_BRP (st_S S1 false B1 B2 B3 B4 L rn) sK sL sR)) ->
                     (X (st_BRP (st_S S1 true B1 B2 B3 B4 sigma rn) sK sL sR)))
            |
            ⋮
            |   ··· }
```

The crucial difference between the types `reach` and `reach_alt` is that the actions are not mentioned explicitly in the definition of `reach_alt`. This is not bothering us when we try to prove `Πx:states_BRP.(reach_alt x)->(INV x)` for some invariant `INV`. In fact, the proof is almost identical to the proof of `Πx:states_BRP.(reach x)->(INV x)`. However, encoding the weak refinement property seems problematic. We cannot use (3) because we don't have the types `step` and `step'` anymore. We could prove

$$\texttt{\Pi s:states\_BRP.(reach\_alt s) -> (reach\_alt' (ref s))} \tag{5}$$

where `reach_alt'` encodes 'reachability' in automaton P. When we prove (5) in a somewhat restricted way then the λ-term, together with the meta argument that this term is of a restricted form, could serve as a proof for the weak refinement property. The restriction is as follows:

Prove `(reach_alt' (ref s))` by a case analysis on the hypothesis `H:(reach_alt s)`. Do not use any other constructors of P then the one corresponding with the subgoal you are working on. (Internal actions of BRP do not correspond with any constructor of P.)

Obviously, this is very inconvenient: It is sometimes difficult to see on which case one is working and hence which hypothesis one may use. Even more unsatisfactory is the fact that the resulting proofterm is incomplete as a justification in itself and needs an additional meta-argument.

4.2 Coq Correctness Proof

Section 4.2.1 will explain the kind of Coq goals that correspond to the proof obligations in this verifications. Section 4.2.2 discusses the use of tacticals in this application.

4.2.1 Goals

Proofs of the invariants and the refinement are essentially by induction over the transitions and split in the corresponding 25 cases (Section 4.1.7). As is to be expected, transitions that do not affect variables that occur in an invariant prove in Coq simply by assumption with the induction hypothesis. Other cases resolve into further subgoals.

In this application of I/O-automata, most predicates are equality assertions over state variables and the proofs involve much propositional reasoning. This is best illustrated by means of an example subgoal: Figure 2 shows a Coq goal that occurs when proving invariant *INVR* (Lemma 3.6). After elimination of reachable states (Section 4.1.7), Coq has filled in the variables and terms in proper places in the states before and after the transition, in the precondition, and in the invariant. The assertion to prove is on top, below that are the assumptions. This case corresponds to action G in case ($ctoggle \rightarrow t{=}toggle$). The latter condition is expressed by assumption H. Other preconditions of this transition arise as equalities over state variables that have been filled in automatically in the states before and after this transition. *H0* and *H1* assume reachability of these states. *H2* contains the induction hypothesis for the invariant property. The goal to prove is that the property holds for states after a G step.

The goal in Figure 2 decomposes in a number of subgoals. Figure 3 focuses on a particular subgoal. The proposition occurs in the rightmost conjunction of the invariant, viz. $R.status{\in}\{\mathsf{RTS},\mathsf{SA}\} \rightarrow R.ftoggle{\neq}R.toggle$. The induction hypothesis (*H2* in Figure 2) has been decomposed into its constituent conjuncts. Applications of projection functions in the goal and in the assumptions have been reduced to retrieve the appropriate terms. *negb* is a function that inverts booleans. *H5* assumes the precondition of this conjunction. The proof uses assumption H.

Note that the statements to prove consist of a logical combination of equality statements. The same observation holds for those assumptions in the context that have not yet been eliminated and can be of relevance to the unfinished proof. This is characteristic of this application and of many I/O-automata proofs. In nearly all cases the equality statements are over finite sets or variables thereover. This holds for preconditions of transition steps as well as for predicates in the invariants. Many proofs are elementary.

Induction serves two purposes in the definition of a set: it states that the given elements are the only inhabitants of the set (*no junk* property) and it states that all elements are different (*no confusion* property). Inductively defined finite sets play an important role in the Coq checking of this verification, both to do analysis by cases as well as to distinguish between elements.

Analysis by cases is provided directly in Coq via elimination of a variable over the inductive set. Inequality of different elements of an inductively defined finite set is not directly available in Coq but must be derived with the *Match* mechanism. Because the verification described here uses this extensively, it will be is illustrated by means of a small example. We inductively define a finite set S and a predicate that discriminates its elements:

```
Inductive Definition S : Set = a : S | b : S | c : S.

Definition neq_S  = [x,y:S](<Prop>Match x with
                              (<Prop>Match y with False True  True)
                              (<Prop>Match y with True  False True)
                              (<Prop>Match y with True  True  False)).
```

```
((<status_R>RTS=NOK)->
  (<bool>(p_ct_R (st_R RTS f l t d B4 B5 B6 B7 B8))=false))
/\(((<status_R>RTS=SI)->
     (<bool>(p_ct_R (st_R RTS f l t d B4 B5 B6 B7 B8))=true)->
      (<bool>(p_ft_R (st_R RTS f l t d B4 B5 B6 B7 B8))
       =(p_t_R (st_R RTS f l t d B4 B5 B6 B7 B8))))
   /\(((<status_R>RTS=RTS)\/(<status_R>RTS=SA))->
       ((<bool>(p_ct_R (st_R RTS f l t d B4 B5 B6 B7 B8))=true)
        /\(<bool>(p_ft_R (st_R RTS f l t d B4 B5 B6 B7 B8))
           =(negb (p_t_R (st_R RTS f l t d B4 B5 B6 B7 B8))))))

============================

  H2 : ((<status_R>WF=NOK)->
         (<bool>(p_ct_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))=false))
       /\(((<status_R>WF=SI)->
            (<bool>(p_ct_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))=true)->
             (<bool>(p_ft_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))
              =(p_t_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))))
          /\(((<status_R>WF=RTS)\/(<status_R>WF=SA))->
              ((<bool>(p_ct_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))=true)
               /\(<bool>(p_ft_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))
                  =(negb (p_t_R (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8))))))

  H1 : (reach
         (st_BRP sS (st_K f l t false d) sL
            (st_R RTS f l t d B4 B5 B6 B7 B8)))

  H0 : (reach
         (st_BRP sS (st_K f l t true d) sL
            (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8)))

  H : ~((<bool>B6=true)->(<bool>B5=t))
  sL : states_L
  sS : states_S
  d1 : data
  B1 B2 B3 B4 B5 B6 B7 B8 : bool
  d : data
  f l t : bool
  S : (step a s1 s2)
  s1 s2 : states_BRP
  a : act_BRP
  R : (reach x)
  x : states_BRP
```

Figure 2: Characteristic Coq subgoal for this application. The assertion to prove is on top, the assumptions are below. The goal forms part of the obligation to prove that transition G preserves invariant *INVR*. *H2* assumes the invariant property holds for states that enable this transition step. The assertion to prove is that the property holds for states after the transition.

```
<bool>t=(negb B5)
============================
  H5 : (<status_R>RTS=RTS)\/(<status_R>RTS=SA)
  H4 : ((<status_R>WF=RTS)\/(<status_R>WF=SA))->
         ((<bool>B6=true)/\(<bool>B3=(negb B5)))
  H3 : (<status_R>WF=SI)->(<bool>B6=true)->(<bool>B3=B5)
  H2 : (<status_R>WF=NOK)->(<bool>B6=false)
  H1 : (reach
          (st_BRP sS (st_K f l t false d) sL
             (st_R RTS f l t d B4 B5 B6 B7 B8)))
  H0 : (reach
          (st_BRP sS (st_K f l t true d) sL
             (st_R WF B1 B2 B3 d1 B4 B5 B6 B7 B8)))
  H  : ~((<bool>B6=true)->(<bool>B5=t))
  sL : states_L
  sS : states_S
  d1 : data
  B1 B2 B3 B4 B5 B6 B7 B8 : bool
  d  : data
  f l t : bool
  S  : (step a s1 s2)
  s1 s2 : states_BRP
  a  : act_BRP
  R  : (reach x)
  x  : states_BRP
```

Figure 3: A subgoal of the goal in Figure 2.

I.e., `(neq_S x y)` reduces to False if $x = y = a$ or $x = y = b$ or $x = y = c$ and it evolves to True otherwise. It serves to prove the desired inequalities $\neg(a = b)$, $\neg(a = c)$, etc. Instead of deriving and naming all n^2 lemmas for an n-ary set, we prove the following generalised lemma to derive contradictions:

$$\forall x, y : S.(< S > x = y) \rightarrow (neq\_Sxy) \rightarrow \forall P : Prop.P$$

Such a lemma is derived for all inductive sets. Say the lemma is named $absurd\_S$. The latter is used extensively to resolve goals with an inconsistent equality assumption in the context, say $< S > a = b$. The following Coq tactical then solves the goal immediately:

`( Apply (absurd_S a b) ; [ Assumption | Simpl ; Exact I ] )` (6)

For invariants that are proved by induction over transition steps, sometimes a majority of the subgoals prove by contradiction because they assume $a = b$ for different a and b from an inductive set.

A small anomaly of Coq is that it cannot distinguish inductive types that have the same structure. Choosing different names for such types just introduces different names for the same basic notion. In particular we cannot distinguish finite sets that have the same cardinality. In this application for instance, `status_S` and `status_R` are two abbreviations for the same type

`Ind(X:Set){ X | X | X | X | X }`. A typing error `<status_R>Rst=WA` in one of the invariants was overlooked for a long time. In this application, one can even prove `<status_R>SF=WF` by reflexivity because `SF` and `WF` are both the first element of a set of five elements (see Section 4.1.5).

4.2.2 Tacticals

Both tactics and tacticals have been used in the proof-checking. Coq tacticals are composed of tactics and they can be used to apply at once a combination of rules. They can also be used to accomplish a limited form of proof search. Such tacticals have been written for five of the invariants in this application. One generic tactical was developed to decompose and investigate versions of $INVR$ (Lemma 3.6), $INVR'$ (Lemma 3.7) and $INVR''$ (Lemma 3.8). After case distinction over the 25 transition steps, the tacticals attempt to decompose these cases by elimination of logical connectives until only simple goals are left, where the assertion to prove is an equality assertion. For our invariants, typically some 50-100 simple goals are left then. Many of these are solved automatically by assumption, reflexivity or by means of an inconsistent equality statement in the context. For the invariants above, only a handful of non-trivial goals remain to be solved by the user.

To achieve a form of search, the tacticals are mainly composed of combinations of the ";" and "*Orelse*" tacticals explained below.

$tactical_1$;
$tactical_2$;
$tactical_3$;
...

This applies $tactical_2$ to the subgoals generated by $tactical_1$ and $tactical_3$ to those that are generated by $tactical_2$.

$tactical_1$ *Orelse*
$tactical_2$ *Orelse*
$tactical_3$

This tries to apply $tactical_1$. If that fails, $tactical_2$ is applied. If that fails, $tactical_3$ is applied. Coq tacticals are fairly basic. A definition mechanism or parameterization is not provided. This would be convenient for this application, since it would allow often recurring tacticals like (6) to be written very compact.

The current Coq tactical language has no variables and pattern matching. As a consequence, tacticals must be tailored to the overall structure of goals if they are used for proof search. Because of this, writing a tactical proof often is as much effort as writing the corresponding tactic proof. Currently, the advantage of such tacticals is mainly that it is easier to adapt them than to adapt tactic proofs: tactical proofs are less affected when invariants or automata are modified.

5 Discussion

The primary objectives of this work have been fulfilled: the protocol has been verified and the verification has been proof-checked. Although the Bounded Retransmission Protocol is small, it is by no means trivial and the efforts involved are considerable. While the PSF specification and simulation activity have been carried out in only two man-weeks, the manual verification took roughly two man-months (including write-up) and the proof-checking took more than three man-months. Part of the latter effort is due to a learning effect. Analysis of the Bounded Retransmission Protocol is not completed: the protocol has an additional *disconnect* service that allows the sender and the receiver to disrupt an ongoing communication. This service has been neglected here and will be verified later.

The verification has answered a number of questions about the protocol. Foremost, it proves that the data link protocol is free of design errors. An important result of the work is that has corrected several inconsistencies, ambiguities, and omissions in the accurate but semi-formal original specification of the protocol. For instance, the exercise has pinned down the behavior of the *toggle* bit between subsequent messages and has formalized many assumptions that were previously left implicit. In addition, the correctness criterion formalizes the required external or black-box behavior of the protocol services.

The automaton specification serves also as a precise functional description for protocol implementations. In this description, all kind of important questions for implementors have been answered, like : "Can I send an empty message?", "How to respond if a request comes before the previous request is completed", "What the start value of the *toggle* bit for subsequent messages?".

Other protocol properties are confirmed by the automaton model. For instance, invariant $INVK'$ (Lemma 3.14) proves that the use of the bit named *first* in data frames is redundant, because the receiver can always predict its value. This is consistent with the situation in the X.25 LAPB protocol [22] that has no comparable field and that uses a *more_data* bit only, which corresponds to the (inverted) bit named *last* in the Bounded Retransmission Protocol. Further, the automaton model confirms that the *first*, *last* and *toggle* bits from the header of acknowledgements are irrelevant for correctness.

The Coq proof-checking confirms that the verification is correct. It was not first-time right though and the proof-checking has corrected a number of draft versions. Both the verification and the specification have been revised several times. Other corrections relate to various errors and inaccuracies in versions of the manuscript proof. Preliminary versions of six invariants required modification. One invariant proved false and required weakening. In four cases the invariants seemed valid but needed strengthening (induction loading) to admit a proof. In one case small modifications to the automaton were necessary to admit a missing proof. Much of the checking was done while parts of the proof were still under development and certain errors must therefore be ascribed to the iterative approach that characterizes the development of automata proofs. Usually the manuscript proof was followed, unless obvious simplifications were seen. For invariant $INVR'$ (Lemma 3.7) the use of tacticals simplified a handwritten proof by abstaining from the application of two other invariants that were used in the manuscript proof.

The experiences with the Coq system are positive. The Coq system 5.8 is robust and reliable and is well-documented. Most shortcomings are related to the ASCII interface: it is easy to lose the overall picture when dealing with large contexts and large proofs.

Modeling the Bounded Retransmission Protocol automata, the invariants and the weak refinement proof in type theory (Coq's Inductive Calculus of Constructions) posed no problem. An important question is if the encoding that has been chosen is satisfactory and how it can be improved upon. A clear advantage of the current mapping to type theory is that it closely follows the application and directly supports the checking of the invariants and the refinement proof. While this encoding thus facilitates the operational checking, it also amalgamates the automata theory and the application which makes it difficult to reuse much of the Coq text for other applications.

An interesting option is to use a more general modeling of automata theory, together with a compact application description similar to the specification in Section 3.2. This can lead to an approach that is more flexible because it allows reuse of the theory modeling for different applications. Also, the simpler application description is less error-prone. The current encoding is tailored towards the proving of invariants and weak refinement relations. Absence of deadlock has to be defined specifically for this application and cannot be reconstructed easily from the transition steps. In an approach that uses a higher level of abstraction, such properties could be defined independent of the particular application. Automatic translation of non-operational application descriptions is desirable. The translation can be within Coq or part of a preprocessor. One may even want to use different translations for different purposes. Some of these options are currently investigated by the authors.

If this application is characteristic of I/O-automata proofs - and this seems to be the case - then I/O-automata verifications could benefit from proof search procedures. Many (sub-)proofs are truly elementary. It must be stressed that this quality does not come for free. In I/O-automata verifications the crucial and most difficult part is *finding* the proper automata, the weak refinement relation and the invariants. This is an iterative process that can benefit from proof search support. The tacticals written for this application indicate that it is feasible to reduce conjectures of invariants to a few non-trivial or impossible subgoals for the user. Proof search can be used in two ways: it can speed up the checking of manuscript proofs but it can also speed up their development. The Coq system is currently designed as a proof-checker and not as a theorem prover. Nevertheless, most proof obligations require very specific and elementary reasoning and some additional tactical building blocks may be of great help for I/O-automata verifications.

Proof-checking of protocol verifications involves choosing a verification formalism and choosing a proof system. In general, the proof system uses a different formalism in which the verification must be embedded.

Several research efforts are aimed at computer support for protocol verifications. A number of example studies are mentioned below, with different verification methods and different proof systems. Engberg, Grønning and Lamport [8] have used the Larch prover (LP) to check verifications in TLA, the Temporal Logic of Actions. The Larch prover is also used by Søgaard-Andersen et.al. [20] to check verifications in I/O-automata. The successful exploitation of a theorem prover confirms our conjecture that proof search may form a useful contribution in the field of I/O-automata theory. Bezem and Groote [3] have used Coq to check a verification of the alternating bit protocol in process algebra. Their proofs are essentially rewriting proofs. Martin Hoffmann [10] in Edinburgh has checked with LEGO a verification of the Alternating Bit Protocol. His verification is based on a functional approach and uses stream transformers.

Acknowledgements

The authors wish to thank Thijs Winter for specifying and simulating the protocol. We also want to thank Jan Friso Groote who verified essentially the same protocol in process algebra [9]. Doeko Bosscher investigated the state space using the Concurrency Workbench [5]. He also corrected an error in an early version of the specification. Final gratitude goes to Rob Jansen and Hans Oerlemans for several discussions on the subject.

References

[1] Baeten, J.C.M. & Weijland, W.P. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[2] Barendregt, H.P. *Lambda Calculi with Types.* In: *Handbook of Logic in Computer Science.* Eds. Abramsky, S., Gabbay, D.M. & Maibaum, T.S.E., pages 117–309. Oxford University Press, 1992.

[3] Bezem, M. & Groote, J.F. *A formal verification of the alternating bit protocol in the calculus of constructions.* Logic Group Preprint Series 88, Dept. of Philosophy, Utrecht University, March 1993.

[4] de Bruijn, N.G. *A survey of the project Automath.* In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalisms.* Eds. Hindley, J.R. & Seldin, J.P., Academic Press, 1980.

[5] Cleaveland, R., Parrow, J. & Steffen, B. *The Concurrency Workbench: a semantics based tool for the verification of concurrent systems.* Technical Report ECS-LFCS-89-83, LFCS, University of Edinburgh, 1989.

[6] Coquand, T. & Huet, G.P. *The Calculus of Constructions.* Information and Computation 76, pp. 95-120, 1985.

[7] Dowek, G., Felty, A., Herbelin, H., Huet, G.P., Murthy, C., Parent, C., Paulin-Mohring, C. & Werner, B. *The Coq proof assistant user's guide. Version 5.8.* Technical report, INRIA – Rocquencourt, May 1993.

[8] Engberg, U., Grønning, P. & Lamport, L. *Mechnical Verification of Concurrent Systems with TLA.* February 1993. To appear in *Proceedings CAV'93.*

[9] Groote, J.F. *A Bounded Retransmission Protocol for Large Data Packets. Logic Group Preprint Series 00.*, Dept. of Philosophy, Utrecht University. To Appear.

[10] Hoffmann, M. *Personal Communication.* 1993.

[11] Jonsson, B. *Compositional Verification of Distributed Systems.* PhD thesis, DoCS 87/09, Department of Computer Systems, Uppsala University, 1987.

[12] Lamport, L. *How to write a proof.* Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993.

[13] Lynch, N.A. & Tuttle, M.R. *Hierarchical correctness proofs for distributed algorithms.* Proc. 6th ACM symposium on Principles of Distributed Computing, pp. 137-151, 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[14] Lynch, N.A. & Tuttle, M.R. *An introduction to input/output automata.* In: *CWI Quarterly*, 2(3):219–246, September 1989.

[15] Lynch, N.A. & Vaandrager, F.W. *Forward and backward simulations for timing-based systems.* In: de Bakker, J.W., Huizing, C., de Roever, W.P. & Rozenberg, G., editors, *Proceedings of the REX Workshop "Real-Time: Theory in Practice"*, volume 600 of *Lecture Notes in Computer Science*, pages 397–446. Springer-Verlag, 1992. Full version available (in two parts) as Reports CS-R9313 and CS-R9314, CWI, Amsterdam, March 1993.

[16] Lynch, N.A. & Vaandrager, F.W. *Forward and backward simulations – part I: Untimed systems.* Report CS-R9313, CWI, Amsterdam, March 1993. Submitted for publication.

[17] Mauw, S. *PSF - A Process Specification Formalism.* Ph.D. thesis, Dept. of Computer Science, University of Amsterdam, 1991.

[18] Paulin-Mohring, C. *Inductive definitions in the system Coq. Rules and properties.* In: Bezem, M. & Groote, J.F., editors, *Proceedings of the* 1^{st} *International Conference on Typed Lambda Calculi and Applications.* Utrecht, The Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

[19] Sellink, M.P.A. *Verifying process algebra proofs in type theory.* Technical Report Logic Group Preprint Series No. 87, Utrecht University, 1993.

[20] Søgaard-Andersen, J.F., Garland, S.J., Guttag, J.V., Lynch N.A. & Pogosyants, A. *Computer-assisted simulation proofs.* February 1993. To appear in *Proceedings CAV'93.*

[21] Tanenbaum, A.S. *Computer Networks*. Prentice Hall, 1989.

[22] CCITT Recommendation X.25. *Interface between DTE and DCE for Terminals Operating in the Packet Mode on Public Data Networks*. CCITT Fascicle VIII.3, 1988.

Elimination of extensionality in Martin-Löf type theory

Martin Hofmann
Department of Computer Science, University of Edinburgh
JCMB, KB, Mayfield Rd., Edinburgh EH9 3JZ, Scotland
e-mail: mxh@dcs.ed.ac.uk

September 1993

Abstract

We introduce axioms of extensionality and quotienting into intensional type theory and show how these can be eliminated using an interpretation of type theory in terms of sets with equivalence relations.

1 Introduction — a discussion of the identity type

Martin-Löf introduced the inductive identity type in order to internalise the notion of definitional equality. For any two terms M, N of some type A a type $\mathrm{Eq}_A(M,N)$ is introduced, which should be inhabited iff M and N are definitionally equal. This is achieved by the following rules.

$$\frac{\Gamma \vdash A}{\Gamma,\ x,y{:}A \vdash \mathrm{Eq}_A(x,y)}$$

Here and in the sequel round brackets with commas are used to denote free variables and substitution in an informal way. So $\mathrm{Eq}_A(x,y)$ could also be written Eq_A if we would not want to emphasize the two particular free variables, and $\mathrm{Eq}_A(M,N)$ is a shorthand for $\mathrm{Eq}_A[x := M][y := N]$. Since we have only one universe, we write $\Gamma \vdash A$ instead of $\Gamma \vdash A\,\mathbf{Set}$. The other rules are:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{refl}_A(M){:}\mathrm{Eq}_A(M,M)}$$

$$\frac{\begin{array}{c}\Gamma,\ x,y{:}A,\ p{:}\mathrm{Eq}_A(x,y) \vdash C(x,y,p)\\ \Gamma,\ x{:}A \vdash M : C(x,x,\mathrm{refl}_A(x))\end{array}}{\Gamma,\ x,y{:}A,\ p{:}\mathrm{Eq}_A(x,y) \vdash \mathrm{J}_C(M) : C(x,y,p)}$$

$$\frac{\begin{array}{c}\Gamma,\ x{:}A,\ p{:}\mathrm{Eq}_A(x,x) \vdash C(x,p)\\ \Gamma,\ x{:}A \vdash M : C(x,\mathrm{refl}_A(x))\end{array}}{\Gamma,\ x{:}A,\ p{:}\mathrm{Eq}_A(x,x) \vdash \mathrm{K}_C(M) : C(x,p)}$$

and equality rules

$$\mathrm{J}_C(M)(x,x,\mathrm{refl}_A(x)) = M(x) \qquad\qquad \mathrm{K}_C(M)(x,\mathrm{refl}_A(x)) = M(x)$$

The eliminator J is the one originally used by Martin-Löf; the homogeneous eliminator K has later been added by Streicher [13] and independently by Altenkirch. One may also devise an η-like equality for the J and K eliminators (cf. [11]). As observed by Streicher [13] this η-rule is equivalent to the equality reflection rule

$$\frac{\Gamma \vdash p : \mathrm{Eq}_A(M, N)}{\Gamma \vdash M = N : A}$$

This rule is known to render definitional equality and type checking undecidable, and was thus rejected by Martin-Löf. In the next paragraph we briefly review the strength and the weaknesses of the identity type without this rule.

This formulation of the identity type captures most of the rules governing definitional equality. Apart from reflexivity, symmetry, and transitivity, in particular the substitution rule for dependent types

$$\frac{\begin{array}{l}\Gamma, x : A \vdash B(x)\\ \Gamma \vdash M = N{:}A\\ \Gamma \vdash U : B(M)\end{array}}{\Gamma \vdash U : B(N)}$$

is reflected in that from J we can define an operator Subst_A, where $\mathrm{Subst}_A(p, U) : B(N)$ if $p : \mathrm{Eq}(M, N)$ and $U : B(M)$ for some type B depending on the type A of M and N. Moreover, by induction, i.e. using J we can prove that these Subst functions are *coherent*, i.e. they do not depend on the proof p and any diagram formed out of these functions commutes up to Eq. Also all the congruence rules for definitional equality hold for the identity type, with the exception of the ξ-rule

$$\frac{\Gamma,\ x{:}A \vdash M = N : B(x)}{\Gamma \vdash \lambda x{:}A.M = \lambda x{:}A.N : \Pi x{:}A.B}$$

which is not provable for the identity type. This means that from a proof of pointwise equality of two dependent functions

$$\Gamma,\ x{:}A \vdash p : \mathrm{Eq}(F, G)$$

we cannot conclude their propositional equality, i.e. we cannot in general find an inhabitant of the type

$$\Gamma \vdash \mathrm{Eq}(\lambda x{:}A.F, \lambda x{:}A.G)$$

The reason for this lies in the equality reflection principle which says that in the empty context an identity type is inhabited if and only if its two arguments are definitionally equal. This follows because the only canonical term of an identity type is refl which only applies in case of definitional equality. Now if the ξ rule held propositionally then two extensionally equal functions on the natural numbers, say, would be conversionally equal, which is in general obviously not the case, for example because definitional equality is decidable and pointwise equality even of primitive recursive functions is not.

So the lack of extensionality can be explained on the grounds that Eq should not identify definitionally different terms. This is, however, a very purist point of view. The real strength of the identity type lies in its substitution property witnessed by Subst. We therefore propose to give up equality reflection, and to understand propositional equality as *substitutability in every context*. Now two pointwise equal functions are indeed substitutable for one another in every context except one arising from the identity type itself. Clearly if $F(x)$ and $G(x)$ are pointwise equal we cannot substitute $\lambda x{:}A.F$ for $\lambda x{:}A.G$ in $\mathrm{Eq}(?, \lambda x{:}A.G)$. The reason is again the lack of extensionality. So we might just add a new constant to the theory

$$\frac{\Gamma, x : \sigma \vdash p : \mathrm{Eq}(F, G)}{\Gamma \vdash \mathrm{Ext}\,(p) : \mathrm{Eq}(\lambda x : \sigma.F, \lambda x : \sigma.G)}$$

which both achieves (via Subst) substitutability of pointwise equal functions in every context, and repairs the just mentioned problem with the particular context $\mathrm{Eq}(\_, G)$. This is essentially the solution proposed by Turner in [14]. Its serious drawback, immediately pointed out by Martin-Löf in a subsequent discussion also in [14] is that this introduces noncanonical elements in each type, since we have not specified, how the eliminator J should behave when applied to Ext . For example, consider the (constant) family $f : A \to B \vdash \mathrm{N}$. Now if $x : A \vdash p : \mathrm{Eq}(Fx, Gx)$ then $\mathrm{Subst}\,(\mathrm{Ext}\,(p), 0)$ is an element of N in the empty context which does not reduce to canonical form. One might try to find suitable reduction rules for Ext under which this term would for example reduce to 0. Yet no satisfactory set of such rules has been found, and we conjecture that there can be no confluent and strongly normalising set of rules which would encompass all the obvious cuts arising from Ext .

The solution to the problem we are going to describe in this paper consists of a post-hoc translation of proofs containing Ext into ones in the pure theory in which every occurrence of the identity type will be replaced by an equality relation defined by induction on the type structure. On basic inductive types this relation will be the identity type itself, but for example on the type $\mathrm{N} \to \mathrm{N}$ it will be pointwise equality and so on.

This translation is performed by constructing a model for type theory including Ext in which types are types with relations and dependent types are dependent types together with dependent relations and substitution functions. The interpretation of the identity type in the model is just the relation associated to each type, which is why extensionality holds in the model.

Given this translation we may view the type theory with Ext as a meta- or macro-language for the theory in which equality is defined along the type structure and in which every substitution must be validated by a tedious proof of substitutivity. Through the interpretation in the model (which can be implemented on a machine) these substitutivity proofs are generated automatically.

For the nondependent case and predicate logic this construction is well-known, in [3] it is attributed to Gandy, cf. also [8]. The idea of interpreting extensional type theory in intensional one was put forward by Martin-Löf during the discussion in [14], but to the best of our knowledge has so far never been made explicit.

2 Quotient types

The model we are going to describe does not only eliminate extensionality, it may also be used to interpret an intensional version of *quotient types* which permit to "factor" types by arbitrary equivalence relations. For lack of space we will in this paper not give the interpretation of quotient types, but only define their syntax, see however Section 10. It is as follows.

$$\frac{\begin{array}{l}\Gamma \vdash A \\ \Gamma,\ x, y{:}A \vdash R\end{array}}{\Gamma \vdash A/R}$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash [M]_R : A/R}$$

$$\frac{\begin{array}{l}\Gamma \vdash M : A \\ \Gamma \vdash N : A \\ \Gamma \vdash p : R(M, N)\end{array}}{\Gamma \vdash \mathrm{Q}_R(p) : \mathrm{Eq}_{A/R}([M]_R, [N]_R)}$$

$$\frac{\begin{array}{l}\Gamma, x{:}A/R \vdash B \\ \Gamma, x{:}A \vdash M : B([x]_R) \\ \Gamma, x, y{:}A, p{:}R(x,y) \vdash P : \mathrm{Eq}(\mathrm{Subst}\,(\mathrm{Q}_R(p), M(x)), M(y))\end{array}}{\Gamma, x{:}A/R \vdash \mathrm{lift}_{R,B}(M, P, x) : B(x)}$$

and equality (computation) rule

$$\mathrm{lift}_{R,B}(M, P, [N]_R) = M(N)$$

These quotient types permit quite comfortable implementations of data structures arising in algebra like integers or rational numbers. The fact that via $\mathrm{Q}_R(\_)$ the user-defined equality for these types becomes substitutive facilitates many proofs. Again the drawback is that $\mathrm{Q}_R(\_)$ introduces noncanonical elements not only in the identity type, but in fact into all other types as can be seen by a similar argument as the one for Ext . In the syntactic model we are going to describe, these quotient types will be translated into their underlying types together with the symmetric, transitive closure of the quotienting relation, and suitable proofs of substitutivity will be generated via the interpretation by induction along the term structure. Observe that the lift-rule only allows the definition of such functions on the quotient which "respect" the relation.

Our definition of quotient types differs from the one described in [2] in that since we do not have the equality reflection rule we need an instance of Subst in order to make the proviso for quotient elimination (in the lift-rule) well-typed.

3 The setoid model

We shall now describe the syntactic model in detail, and show that its structure suffices to interpret all of intensional type theory including Ext . The underlying syntactic system is intensional Martin-Löf type theory without universes as described in [10]. Our presentation is clearly influenced by "categorical type theory" as described e.g. in [6, 12], we shall, however, avoid categorical terminology as far as possible and we assume no knowledge of category theory. For the *cognoscenti*: we construct a "category with attributes" in the sense of Cartmell [1], see also [7].

We begin the construction with nondependent sets with relations which will later serve to interpret the contexts.

Definition 1 (Setoids) *A* setoid[1] *consists of a type in the empty context*

$$\vdash X$$

[1]This terminology is due to Rod Burstall.

and a relation on it

$$x, y{:}X \vdash R(x, y)$$

which is provably an equivalence relation, i.e. there are terms

$$\begin{array}{l} x{:}X \vdash \mathit{refl}(x) : R(x,x) \\ x,y{:}X,\ p{:}R(x,y) \vdash \mathit{sym}(p) : R(y,x) \\ x,y,z{:}X,\ p{:}R(x,y),\ q{:}R(y,z) \vdash \mathit{trans}(p,q) : R(x,z) \end{array}$$

If X is a setoid then we refer to its components by X_{set}, X_{rel}, X_{refl}, X_{sym}, X_{trans}, resp. Two setoids are equal if all their five components are equal.

Every type X gives rise to a setoid by taking the identity type as the relation which is an equivalence relation. One can also see that the cartesian product (degenerated Σ-type) of setoids is a setoid again. In particular the empty product, i.e. the unit type is a setoid. We shall use it to interpret the empty context.

Two general comments concerning the syntactic nature of setoids are in order. First, by a setoid we really mean the syntactic object itself consisting of several types, terms, and judgements; and not its interpretation in some model. Second, the "proofs" of reflexivity, symmetry, etc. are not merely required to exist, but form an intrinsic part of a setoid. This also applies to the other syntactic objects we are going to define.

Definition 2 (Morphisms of setoids) *A morphism between two setoids X and Y consists of a "function"*

$$x{:}X_{\mathrm{set}} \vdash \mathit{fun}(x) : Y_{\mathrm{set}}$$

and a proof that it respects the relations, i.e. a term

$$x, y{:}X_{\mathrm{set}},\ p : X_{\mathrm{rel}}(x, y) \vdash \mathit{resp}(p) : Y_{\mathrm{rel}}(\mathit{fun}(x), \mathit{fun}(y))$$

Again, if F is a morphism of setoids we refer to its components by F_{fun} and F_{resp}. Two morphisms of setoids are equal if both components are equal. The set of setoid morphisms from X to Y is denoted by $\mathrm{Mor}\,(X, Y)$.

The reader is invited to check that the projections, as well as the pairing functions corresponding to a cartesian product form morphisms of setoids. Moreover, one can easily see that morphisms of setoids contain the identities and are closed under composition and thus form a category. In fact this category is cartesian closed, and although we will not need it for the interpretation we give the construction of the exponential, since its construction is similar to the one of the dependent product we need later. So let X and Y be setoids. Their exponential $X \Rightarrow Y$ is defined by

$$(X \Rightarrow Y)_{\mathrm{set}} := \Sigma f{:}X_{\mathrm{set}} \to Y_{\mathrm{set}}.\Pi x, y{:}X_{\mathrm{set}}.X_{\mathrm{rel}}(x, y) \to Y_{\mathrm{rel}}(f\ x, f\ y)$$

$$(X \Rightarrow Y)_{\mathrm{rel}}[f, g : (X \Rightarrow Y)_{\mathrm{set}}] := \Pi x, y{:}X_{\mathrm{set}} \to Y_{\mathrm{set}}.\Pi x, y{:}X_{\mathrm{set}}.X_{\mathrm{rel}}(x, y) \to Y_{\mathrm{rel}}(f.1\, x, g.1\, y)$$

Here .1 and .2 denote the projections of the Σ-type and the $[\ldots]$ notation facilitates the definition of terms with free variables in a hopefully understandable way. We leave it as an exercise to define the remaining components and to verify that this defines indeed the desired exponential. It is important that the underlying set of the exponential is not the full function space, since otherwise the relation could not be proven reflexive. In fact on higher types reflexivity actually means substitutivity, and it is by the requirement that all relations be reflexive that we achieve the interpretation of the substitutive identity type.

4 Families of setoids

The most important ingredient needed to interpret Martin-Löf type theory is the correct definition of type dependency. So we must say what a family of setoids indexed over a setoid is.

Definition 3 *Let X be a setoid. A* family of setoids indexed over X *is given by the following data.*

- *A type depending on X_{set}*

$$x{:}X_{\mathrm{set}} \vdash Y(x)$$

- *A dependent relation on Y*

$$x, x'{:}X_{\mathrm{set}},\ y{:}Y(x),\ y'{:}Y(x') \vdash S(y, y')$$

- *A "reindexer" which allows to substitute related elements of X_{set} in Y*

$$x, x'{:}X_{\mathrm{set}},\ p{:}X_{\mathrm{rel}}(x, x'),\ y{:}Y(x) \vdash F(p, y){:}Y(x')$$

such that S is an equivalence relation and such that $F(y)$ is always S-related to y. More precisely we require terms

$$x{:}X_{\mathrm{set}},\ y{:}Y(x) \vdash \mathit{refl}(y) : S(y, y)$$

$$x, x'{:}X_{\mathrm{set}},\ p{:}X_{\mathrm{rel}}(x, x'),\ y{:}Y(x),\ y'{:}Y(x'),\ q{:}S(y, y') \vdash \mathit{sym}(p, q) : S(y', y)$$

$$\begin{array}{l} x, x', x''{:}X,\ p{:}X_{\mathrm{rel}}(x, x'),\ p'{:}X_{\mathrm{rel}}(x', x''), \\ y{:}Y(x),\ y'{:}Y(x'),\ y''{:}Y(x''),\ q{:}S(y, y'),\ q'{:}S(y', y'') \vdash \mathit{trans}(p, p', q, q') : S(y, y'') \end{array}$$

$$x, x'{:}X_{\mathrm{set}},\ p{:}X_{\mathrm{rel}}(x, x'),\ y{:}Y(x) \vdash \mathit{ax} : S(y, F(p, y)))$$

If Y is a family of setoids indexed over X we shall denote its ingredients by Y_{set}, Y_{rel}, Y_{reindex}, ... resp.

The idea behind the rewriter F is to allow substitution inside a dependent family if the indexing elements are "equal", i.e related. It will be the main ingredient in the interpretation of the identity elimination rule in the setoid model.

If X and Y are setoids we can form a constant family of setoids indexed over X whose underlying set and relation are just their weakened companions taken from Y, whereas the reindexer is the identity. Every ordinary dependent type induces a family of setoids with the identity type as relation, and Subst_A as reindexer. More examples arise from the constructions on families of setoids we are going to describe.

5 Context comprehension

In the setoid model contexts will be interpreted as setoids, whereas types (in contexts) will be interpreted as families of setoids indexed over their context. So the first thing we have to define is context comprehension, i.e. the rule

$$\frac{\Gamma \vdash A}{\Gamma, x{:}A \vdash}$$

The judgement $\Gamma \vdash$ means that Γ is a derivable context. Let Y be a family of setoids indexed over X. Its comprehension denoted $\mathrm{Compr}\,(X,Y)$ is the setoid having as underlying set the dependent sum $\Sigma x : X_{\mathrm{set}}.Y_{\mathrm{set}}(x)$ and as relation

$$\mathrm{Compr}\,(X,Y)_{\mathrm{rel}}[u,v : \mathrm{Compr}\,(X,Y)_{\mathrm{set}}] = X_{\mathrm{rel}}(u.1,v.1) \times Y_{\mathrm{rel}}(u.2,v.2)$$

where $\times$ is shorthand for the nondependent special case of the Σ-type. We have omitted the first two variables to Y_{rel} since they can be inferred from the context. Where appropriate we shall do that in the sequel as well. That this is an equivalence relation is an easy consequence of the laws for X_{rel} and Y_{rel}. Indeed the axioms on Y_{rel} were precisely chosen so as to make $\mathrm{Compr}\,(X,Y)_{\mathrm{rel}}$ an equivalence relation.

We must also interpret the canonical projection from the enlarged context to the original one. We define $\mathrm{compr}\,(X,Y) \in \mathrm{Mor}\,(\mathrm{Compr}\,(X,Y),X)$ by

$$\mathrm{compr}\,(X,Y)_{\mathrm{fun}}[u{:}\mathrm{Compr}\,(X,Y)_{\mathrm{set}}] := u.1$$

$$\mathrm{compr}\,(X,Y)_{\mathrm{resp}}[u,v{:}\mathrm{Compr}\,(X,Y)_{\mathrm{set}},\ p{:}\mathrm{Compr}\,(X,Y)_{\mathrm{rel}}(u,v)] := p.1$$

6 Sections of families

Instead of defining arbitrary morphisms between families we restrict ourselves to "sections" which will be used to interpret terms and can be viewed as family morphisms from the constant (unit) family corresponding to the context itself into a family.

Definition 4 (Sections) *If Y is a family over a setoid X then a section of Y consists of a term*

$$x{:}X_{\mathrm{set}} \vdash el : Y_{\mathrm{set}}(x)$$

and a proof that it respects the relations

$$x,x'{:}X_{\mathrm{set}},\ p{:}X_{\mathrm{rel}}(x,x') \vdash resp(p) : Y_{\mathrm{rel}}(x,x',el(x),el(x'))$$

We denote the set of sections of Y by $\mathrm{Sect}\,(Y)$ and refer to the components of a setoid by $\_{\mathrm{el}}$ and $\_{\mathrm{resp}}$.

Every section induces a setoid morphism from its context to the comprehension of its type, more precisely if $M \in \mathrm{Sect}\,(Y)$ then we get a morphism in $\mathrm{Mor}\,(X,\mathrm{Compr}\,(X,Y))$ whose function part is given by

$$\lambda x{:}X_{\mathrm{set}}.(x,M_{\mathrm{el}}(x))$$

In the sequel we shall identify a section with its associated morphism

7 Weakening and substitution

Instead of defining the substitution[2] of an element (a section) into a family we define substitution for arbitrary setoid morphisms which gives both simultaneous substitution by several terms and weakening as special cases. This technique is reminiscent from categorical models of type theory. Under this interpretation it is helpful to think of setoid morphisms as of tuples of terms ("context

[2]We apologise for the apparent overloading of the term "substitution".

morphisms" or "substitutions"). So let S be a family of setoids indexed over Y and f be a setoid morphism from X to Y. We obtain a family of setoids over X denoted $S[f]$ by putting

$$S[f]_{\mathrm{set}}[x{:}X_{\mathrm{set}}] := S_{\mathrm{set}}(f_{\mathrm{fun}}(x))$$

$$S[f]_{\mathrm{rel}}[x,x'{:}X_{\mathrm{set}},\ y{:}S_{\mathrm{set}}(f_{\mathrm{fun}}(x)),\ y'{:}S_{\mathrm{set}}(f_{\mathrm{fun}}(x'))] := S_{\mathrm{rel}}(f_{\mathrm{fun}}(x), f_{\mathrm{fun}}(x'), y, y')$$

$$S[f]_{\mathrm{reindex}}[x,x'{:}X_{\mathrm{set}},\ p{:}X_{\mathrm{rel}}(x,x'),\ y{:}S_{\mathrm{set}}(f_{\mathrm{fun}}(x))] := S_{\mathrm{reindex}}(f_{\mathrm{resp}}(p), y)$$

We leave out the obvious proof components. This substitution operation is the first example of a construction on setoids which blurs the distinction between "computational" part and "proof" part, since the second component of the morphism f becomes part of the rewriter for the substituted family. Later on, when we shall define the identity type for setoids this distinction will be destroyed completely.

Substitution along a morphism arising from a section corresponds to real substitution as in

$$\frac{\Gamma,\ x{:}A \vdash B(x)}{\Gamma \vdash B(M)}$$

whereas substitution along a morphism $\mathrm{compr}\,(X,Y)$ interprets weakening

$$\frac{\Gamma \vdash B}{\Gamma,\ x{:}A \vdash B}$$

If no confusion can arise we abbreviate $T[\mathrm{compr}\,(X,S)]$ simply by T^{+}. Remember also that since we have notationally identified sections and the corresponding morphisms, the substitution of $M \in \mathrm{Sect}\,(X,S)$ into T above $\mathrm{Compr}\,(X,S)$ will simply be written as $T[M]$.

Substitution also applies to sections; if $M \in \mathrm{Sect}\,(S)$ then we obtain a section $M[f] \in \mathrm{Sect}\,(S[f])$ in the straightforward way. There is also a context morphism arising from substitution which is a bit difficult to understand at first. It goes from $\mathrm{Compr}\,(X,S[f])$ to $\mathrm{Compr}\,(Y,S)$ and we denote it by $q(f,S)$. Its function component is defined by

$$q(f,S)_{\mathrm{fun}}[u : \Sigma x{:}X_{\mathrm{set}}.S_{\mathrm{set}}(f_{\mathrm{fun}}(x))] := (f_{\mathrm{fun}}(u.1),\ u.2) : \Sigma y{:}Y_{\mathrm{set}}.S_{\mathrm{set}}(y) : \mathrm{Compr}\,(Y,S)_{\mathrm{set}}$$

It is used to perform substitutions in variables other than the last one like in

$$\frac{\begin{array}{l}\Gamma \vdash A\\ \Gamma,\ x{:}A \vdash B\\ \Gamma,\ x{:}A,\ y{:}B \vdash C\\ \Gamma \vdash M : A\end{array}}{\Gamma,\ y{:}B(M) \vdash C(M,y)}$$

Here A is a family of setoids over Γ, B is one over $\mathrm{Compr}\,(\Gamma,A)$, and C is a family over $\mathrm{Compr}\,(\mathrm{Compr}\,(\Gamma,A),B)$. M is a section of A. The conclusion is then obtained as

$$C[q(M,B)]$$

It is a characteristic property of substitution that the square of morphisms f, $\mathrm{compr}\,(Y,S)$, $\mathrm{compr}\,(X,S[f])$, $q(f,S)$ is a pullback.

The final ingredient we require to interpret all manoeuvres with variables and substitutions is the last variable in a context, i.e.

$$\Gamma,\ x{:}A \vdash x : A$$

In our setup this is a section of A^+, since the type A on the rhs is actually weakened. We omit the definition of this section which we denote by $\mathrm{v0}\,(A)$.

It should be stressed that our substitution inherits the split property from the syntax. This means that for $f \in \mathrm{Mor}\,(X, Y)$ and $g \in \mathrm{Mor}\,(Y, Z)$ and S above Z we have the type equality

$$S[g][f] = S[g \leq f]$$

and also

$$S[\mathrm{id}_Z] = S$$

where id_Z is the identity morphism and $\leq$ is composition in the applicative order.

8 Dependent product

Let G be a setoid, S be a family over G and T be a family over $\mathrm{Compr}\,(G, S)$. We want to define a family over G which "internalises" the sections of T, i.e. the dependent product $\Pi(S, T)$. As in the case of the exponential of setoids its underlying set is a Σ-type

$$\Pi(S,T)_{\mathrm{set}}[g : G_{\mathrm{set}}] := \Sigma F : \Pi s{:}S_{\mathrm{set}}(g).T_{\mathrm{set}}(g,s)\ .\ \Pi s,s'{:}S_{\mathrm{set}}(g).S_{\mathrm{rel}}(s,s') \to T_{\mathrm{rel}}(F\ s, F\ s')$$

The relation is defined accordingly by

$$\begin{aligned}\Pi(S,T)_{\mathrm{rel}}[g,g'{:}G_{\mathrm{set}},\ U{:}\Pi(S,T)_{\mathrm{set}}(g),\ U{:}\Pi(S,T)_{\mathrm{set}}(g')] :=\\ \Pi s{:}S_{\mathrm{set}}(g)\Pi s'{:}S_{\mathrm{set}}(g').S_{\mathrm{rel}}(s,s') \to T_{\mathrm{rel}}(U.1\ s, V.1\ s')\end{aligned}$$

We leave out the definition of the other components, but mention that the proof of transitivity requires the rewriter S_{reindex}.[3].

Next we define introduction and elimination for the dependent product, more precisely if $M \in \mathrm{Sect}\,(T)$ we construct

$$\Pi_{\mathrm{intro}}\,(S, T, M) \in \mathrm{Sect}\,(\Pi(S, T))$$

and conversely if $M \in \mathrm{Sect}\,(\Pi(S, T))$ and $N \in \mathrm{Sect}\,(S)$ we construct

$$\Pi_{\mathrm{elim}}\,(S, T, M, N) \in \mathrm{Sect}\,(T[N])$$

in the straightforward way.

[3]This means that if we had defined families of setoids without the rewriter, it would have been impossible to define the dependent product. This also means that if we apply the setoid construction to an arbitrary locally cartesian closed category (lccc) we do not obtain an lccc again, but only a "display map category" [12] the display maps being those maps which have a rewriter. In categorical terms these are the *fibrations* in the 2-category of setoids. In the author's opinion this fact is a convincing argument as to why locally cartesian closed categories cannot be considered as a general notion of model for type theory.

Compatibility with substitution Since in a model substitution is given as an additional operation and cannot be defined by induction, it requires a proof that the inductive substitution laws are actually satisfied. In other words we have to show that substitution commutes with product formation, as well as with introduction and elimination. In categorical jargon this is referred to as "Beck-Chevalley condition". Consider the following situation, G, S, T as before; B a setoid and f a setoid morphism from B to G. Now we may form the product of T and then substitute along f:

$$\Pi(S,T)[f]$$

or first substitute f into both S and T and then take the product:

$$\Pi(S[f],\ T[q(f,S)])$$

Unfortunately these families do *not* agree. Their sets of sections are equal, however, so that we do not have to introduce explicit conversion functions to pass from one type to the other. There is no problem with the introduction and elimination operators, they commute with substitution.

Equality rules It remains to check the β and η-equations for Π-introduction and elimination. Both are inherited from their syntactic companions.

9 The identity type

We can now reap the fruits of the laborious model constructions carried out so far and define the identity setoid which will satisfy the extensionality principle. Suppose we are given a setoid G and a family of setoids S over G. We first form the context consisting of G and two copies of S. In combinator notation this is $\mathrm{Compr}\,(\mathrm{Compr}\,(G,S),\ S^+) =: G.S.S$. The identity setoid denoted $\mathrm{Eq}\,(S)$ is a family over this. We define its components in order.

$$\mathrm{Eq}\,(S)_{\mathrm{set}}[(g,s_1,s_2) : (G.S.S)_{\mathrm{set}}] := S_{\mathrm{rel}}(g,g,s_1,s_2)$$

Notice that the underlying set of the context for Eq is a Σ-type with three components.

$$\mathrm{Eq}\,(S)_{\mathrm{rel}}[u,u'{:}(G.S.S)_{\mathrm{set}},\ i{:}\mathrm{Eq}\,(S)_{\mathrm{set}}(u),\ i'{:}\mathrm{Eq}\,(S)_{\mathrm{set}}(u')] := \mathbf{1}$$

where $\mathbf{1}$ is the unit type with single inhabitant $\star : \mathbf{1}$. So all elements of the identity setoid are related. This is clearly an equivalence relation. To define the rewriter $\mathrm{Eq}\,(S)_{\mathrm{reindex}}$ we make use of transitivity and symmetry of S_{rel}. We shall now embark on the definition of the combinators associated with the identity type. We henceforth use the informally introduced dot notation for successive context comprehensions.

Reflexivity The canonical element of the identity type is a section of $\mathrm{Eq}\,(S)[\mathrm{v0}\,(S)]$. It is defined by

$$\mathrm{refl}\,(S)_{\mathrm{el}}[(g,s) : \mathrm{Compr}\,(G,S)] := S_{\mathrm{refl}}(g,s)$$

The $_{\mathrm{resp}}$ component is trivial since any two elements of the identity setoid are related.

Identity elimination The main part in the definition of the J-eliminator for $\mathrm{Eq}\,(S)$ is the rewriter S_{reindex}. We must do a bit of work though in order to get the various contexts and substitutions right. Let C be a family of setoids indexed over $G.S.S.\mathrm{Eq}\,(S)$. Substituting refl into it gives

$$C[q(\mathrm{v0}\,(S), \mathrm{Eq}\,(S))][\mathrm{refl}\,(S)]$$

Let M be a section of this family which we will abbreviate by $C(\mathrm{refl})$. We must construct a section of C from this. We define

$$\begin{aligned}&\mathrm{J}(C,M)_{\mathrm{el}}[(g,s_1,s_2,p) : G.S.S.\mathrm{Eq}\,(S)] := \\ &\quad C_{\mathrm{reindex}}((G_{\mathrm{refl}}(g),\ S_{\mathrm{refl}}(s_1),\ p,\ \star),\ (M_{\mathrm{el}}(g,s_1)))\end{aligned}$$

We omit the $\__{\mathrm{resp}}$ part of $\mathrm{J}(C,M)$. Its main ingredient is S_{ax} — the proof that rewriting does not alter the S_{rel}-class. The definition of the other eliminator K is similar.

Unfortunately our definition of identity elimination does not validate the corresponding β-rules up to conversion, i.e. we do not have

$$\mathrm{J}(C,M))[q(\mathrm{v0}\,(S), \mathrm{Eq}\,(S))][\mathrm{refl}\,(S)] = M$$

We can, however, show that the corresponding identity type is inhabited, i.e. that both sides are related. So in the interpretation we have to replace all instances of the β-rule for equality by propositional equalities and suitably interspersed Subst -operations. Indeed one might envisage a completely descriptive type theory with equality confined to syntactic identity and computation rules replaced by propositional equalities. Terms are then in 1-1 correspondence to derivations. Fortunately in our model computation for the Π-type does hold and it is only in the case of the identity type that we have to accept this less comfortable presentation.

Extensionality Let G and S be as before, T above $G.S$ and U, V sections of T. Moreover, assume a section M of $\mathrm{Eq}\,(T)[V[\mathrm{compr}\,(G.S,T)]][U] =: \mathrm{Eq}\,(S)(U,V)$, i.e. a proof that U and V are equal. We must show that their respective abstractions are equal as well — we need a section of

$$\mathrm{Eq}\,(\Pi(S,T))[\Pi_{\mathrm{intro}}\,(T,V)^{+}][\Pi_{\mathrm{intro}}\,(T,U)]$$

The element part of M is basically (modulo some currying of Σ-types) a term of type

$$g{:}G_{\mathrm{set}},\ s{:}S_{\mathrm{set}}(g) \vdash S_{\mathrm{rel}}(U_{\mathrm{el}}(g,s),\ V_{\mathrm{el}}(g,s))$$

The element we are looking for amounts to an inhabitant of

$$g{:}G_{\mathrm{set}},\ s,s'{:}S_{\mathrm{set}}(g) \vdash S_{\mathrm{rel}}(U_{\mathrm{el}}(g,s),\ V_{\mathrm{el}}(g,s'))$$

We obtain that by using either U_{resp} or V_{resp} and transitivity. The $\__{\mathrm{resp}}$ part is again trivial.

Compatibility with substitution Fortunately with the identity type no such problems as with the dependent product arise, all constructions commute with substitution up to conversion. However the identity types of the two different but should-be equal instances of a substituted Π remain different. Again, however, it is possible to pass from one to the other in a coherent way.

10 Interpretation of other type constructors

Inductive types like the natural numbers or the booleans lift straightforwardly to the setoid model; the relation is simply the identity type. At the moment we are not sure about parametrised inductive types like lists or trees; we expect, however, that a suitable relation can always be formulated.

Strong Σ-types can be interpreted staightforwardly, we have done so in the Lego implementation mentioned below.

It should be possible to interpret a universe by using the identity type as relation, but the details remain to be checked.

A quotient type will have as underlying set the same as the one of its type of representatives. The relation, however, will be the symmetric, transitive closure of the quotienting relation. The fact that this relation is given "internally" as a family of setoids means that it is compatible with the relation on the set of representatives. We will describe this more precisely and more formally in a future paper.

11 Other approaches to extensionality

There are at least two more solutions to the extensionality problem. One consists of adding a new universe of propositions which does not affect the types at all. One is then free to add any propositional assumptions one likes, provided one can give a model or other proof of consistency, but one loses the possibility of extracting programs from proofs, in particular the possibility to reindex dependent types along propositional equalities. If one is interested in formalisations of algebra where dependent types play a subordinate rôle, this is a very simple yet sound approach.

The other solution is again a syntactic model construction, which differs from the given one in that we interpret families of setoids as nondependent sets where a non-reflexive relation singles out the different fibres. The identity type becomes the unit type under this interpretation. In this model the type part and the relation part are completely separated, so that the relation part can be projected away. What remains is a kind of realisability interpretation of type theory. Dependent types are interpreted as simple types, dependent product as rrow type and the identity type becomes the unit. Both constructions will be described in the author's forthcoming thesis [4].

12 Lego implementation

The setoid model has been implemented in the Lego system [9]. This means that the combinators we have defined are actually available and setoids can be built together using these combinators according to some derivation in type theory. Also we have used Lego to check all the equations which are required to hold. For the future it might be useful to have an interpreter which translates actual lambda terms into such combinators.

13 Conclusion

We have given an interpretation of intensional type theory with the Ext-axiom in intensional type theory. One may now use the type theory with Ext and eliminate its occurrences by a post-hoc interpretation in the setoid model. But is there any use in doing so? Certainly the terms obtained from this interpretation can be executed, whereas the terms containing Ext cannot. But if one is merely interested in execution one may just as well use the much simpler realisability interpretation into untyped lambda calculus where Ext can be realised by the identity.

In our opinion the real value of model constructions like the one we have described lies in the syntactic justification of rules like Ext or quotient types, and in the insight that formal proof development in type theory does not necessarily mean to carry around lots of different equalities and substitutivity proofs, since in principle the translation into setoids may always be performed.

¿From a theoretical point of view our construction is interesting since it represents an alternative to the approach to quotients described e.g. in [5] where morphisms are functional relations instead of true functions and thus correspond to specifications rather than actual algorithms. From outside the main difference is that our quotient types are not *effective*. This means that even if R is an equivalence relation it is in general impossible to conclude from $\mathrm{Eq}_{A/R}([Q]_R, [N]_R)$ that M and N are actually related.

References

[1] J. Cartmell. *Generalized algebraic theories and contextual categories.* PhD thesis, Univ. Oxford, 1978.

[2] Robert Constable and D. J. Howe. Nuprl as a general logic. In *Logic and Computer Science*, pages 123–196. Academic Press, 1990.

[3] Solomon Feferman. Theories of Finite Type. In Jon Barwise, editor, *Handbook of Mathematical Logic*, chapter D.4, pages 934–935. North-Holland, 1977.

[4] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, Univ. of Edinburgh, forthcoming 1994.

[5] J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts. Tripos theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 88:205–232, 1980.

[6] Bart Jacobs. *Categorical Type Theory.* PhD thesis, University of Utrecht, 1991.

[7] Bart Jacobs. Comprehension categories and the semantics of type theory. *Theoretical Computer Science*, 107:169–207, 1993.

[8] Horst Luckhardt. *Extensional Gödel Functional Interpretation. A Consistency Proof of Classical Analysis*, volume 306 of *Lecture Notes in Mathematics.* Springer, Berlin, 1973.

[9] Zhaohui Luo and Randy Pollack. LEGO Proof Development System: User's Manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[10] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis·Napoli, 1984.

[11] Robert A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, 95:33–48, 1984.

[12] Thomas Streicher. *Semantics of Type Theory.* Birkhäuser, 1991.

[13] Thomas Streicher. *Semantical Investigations into Intensional Type Theory.* Habilitationsschrift, LMU München, to appear 1993.

[14] David Turner. A new formulation of constructive type theory. In P. Dybjer, editor, *Proceedings of the Workshop on Programming Logic*, pages 258–294. Programming Methodology Group, Univ. of Göteborg, May 1989.

Refinement and local undo in the interactive proof editor ALF

Lena Magnusson[§]
Department of Computer Science,
University of Göteborg/Chalmers

Preliminary version, September 1993

Abstract

ALF is a proof editor in which the proofterm plays an important role. To prove a statement in ALF is to interactively construct a proofterm by direct manipulation of the (partial) proofterm. A partial proofterm is a term which contains *placeholders*, i.e. "holes" which are meant to be filled in. The two main operations on a proofterm is to replace a placeholder by a partial proofterm (refinement), and to replace a partial proofterm by a placeholder (local undo). Local undo refers to a "local" removal of the unwanted part of a proofterm, in contrary to the the global undo (or state undo) which acts on the global state of the proofsystem. Refinement and local undo are dual actions in the sense that the state of ALF is determined entirely by the partial proofterm, regardless of how and in what order the two operations has been used to build that proofterm. To achieve duality between the operations, information about dependencies and sharing must be taken into account in the representation of partial proofterms. The proofterm representation and the procedures of the two operations are explained in this paper.

1 Introduction

The aim of ALF is to have a proof editor in which formal proofs can be made as convenient and flexible as possible. The object to be edited in ALF is a (named) partial proof term, in contrary to most other systems which are command oriented (and acts on the state of the proof engine). A proofterm may contain several placeholders, and the user may choose to refine any placeholder at any time. Moreover, there may be several partial proofterms simultaneously, which for example allows the user to add a lemma at any time. The user chooses freely which proofterm to work on. The flexibility in construction order makes it possible to fully take advantage of the typechecking, since refining a placeholder may force the instantiation of another placeholder of which the first depends. The instantiation is done by unification, but since placeholders may be of higher order and higher order unification is neither complete nor decidable [Hue75], unification can not always solve the equations restricting the placeholders. We have chosen to leave the "difficult part" of unification as *constraints*, and the "simple part" will be automatically instantiated.

[§]lena@cs.chalmers.se

Even though *constructing* proofs is the purpose of these systems, we believe that some kind of undo mechanism is an important feature of a practical system. The obvious reason is that if we seriously want to construct proofs in these systems, and not simply check formally completed proofs, there must be a convenient way to recover from mistakes. Most systems in the same class as ALF support the global undo mechanism (if they support backtracking at all), which goes well together with state transition systems. However, we believe that with the flexibility of manipulating proofterms rather freely, we also need another way of erasing wrong attempts. For example, global (state) undo could force the deletion of a completely unrelated part of the proofterm, or even another proofterm, and an extensive amount of work may be lost. With the local undo mechanism, the user can remove any part (and only that part) of a proofterm he/she desires. It should be noted that this may resume in a completely new state, which is not possible with the state undo operation. The desire for local undo is also expressed in [TBK92]. On the other hand, if we restrict ourselves to constructing proofterms strictly in dependency order (left to right, top to bottom), global undo would suffice, since the two undo operations would have the same effect.

There are possibly some additional usage of the local undo operation. In programming one often reuses code by copying parts of the program text and makes (small) alterations to get a similar program. Copying and using local undo might be a convenient way of producing proofs similar to completed proofs. Another interesting aspect to investigate is the possibility of using the old ancient way of solving problems by example (see [Mäe93]) and generalize the solution by local undo.

This paper is organized in the following way, first we will give an example as motivation for our local undo, followed by an explanation of the problems with the operation. Then we describe the *scratch area* which is the proof construction part of ALF. The part concerning the theory definitions (the environment) is beyond the subject of this paper, but can be found in [Mag92]. We will continue by describing the *constraints*, which are the equations restricting the choices of unknowns, and the procedure of simplifying a set of constraints. Section 8 and 9 explains the two main operations on the scratch area, namely refinement of an unknown and local undo. Finally, some optimizations in the actual implementation of the algorithms presented.

2 Motivation of local undo

We believe that the advantage of local undo is well illustrated in the little story below (see picture 1.) The story is about Calvin who is getting dressed for school one winter morning.

The story can easily be formalized in ALF. We will define the set of clothes by

$Clothes : Set$
- $hat : Clothes$
- $scarf : Clothes$
- $jacket : Clothes$
- $socks : Clothes$
- $shoes : Clothes$
- $mittens : Clothes$

and to be dressed as a proposition which says that there exists a list of clothes such that there

Figure 4: Local undo beats global undo...

is one of each of the clothes items and they are in the proper order:

$Dressed \equiv \exists l \in List(Clothes).OneOfEach(l) \& ProperOrder(l)$

where *OneOfEach* is defined as

$OneOfEach(l) \equiv (lenght(l) =_N 6) \;\&\; Mem(hat, l) \;\&\; Mem(scarf, l) \;\&\ldots\&\; Mem(mittens, l)$

and *ProperOrder* as

$ProperOrder(l) \equiv$

$(last(l) =_{Clothes} mittens) \;\&\; Before(socks, shoes, l) \;\&\; Before(jacket, scarf, l)$

since we all know that it is impossible to get dressed with big mittens on, that socks must be put on before shoes and jackets must be put on before scarfs. *Before* is inductively defined with two constructors,

Before : (c1,c2: Clothes ; l: List(Clothes)Set

before1 : (c1,c2: Clothes ; l : List(Clothes); h: Mem(c2,l)) Before(c1,c2,c1::l)

before2 : (c1,c2,c3: Clothes ; l:List(Clothes); h:Before(c2,c3,l)) Before(c2,c3,c1::l) where the first constructor states that if $c1$ is the first element in a list and $c2$ is a member of that list, then $c1$ is before $c2$ in that list. The second constructors corresponds to the inductive case, when we know that $c2$ is before $c3$ in a list it still holds if we put on element in front of the list.

Now Calvin has to prove that he is properly dressed. He proceeds by finding the six items of clothes which corresponds to refining the unknown goal with a list of length six
$l = [?_1, ?_2, ?_3, ?_4, ?_5, ?_6]$
and he can prove that $length(l) =_N 6$. He puts on his shoes, by refining $?_1 = shoes$
$l = [shoes, ?_2, ?_3, ?_4, ?_5, ?_6]$
and continues with the jacket, scarf, hat and mittens, resulting with the list
$l = [shoes, jacket, scarf, hat, mittens, ?_6]$
and he can simultaneously prove $Mem(shoes, l)$, $Mem(jacket, l)$ $, \ldots,$ $Mem(mittens, l)$ and $Before(jacket, scarf, l)$. But at this point he realizes that his socks is left over... He tries to sneak out the door, but Calvins mom (read ALF) forbids it. To prove *OneOfEach(l)* he needs $?_6 = socks$, but this violates that *last(l)* $=_{Clothes}$ *mittens*. This is a dead end. After some time of deep contemplation, Calvin realizes that he must not get totally undressed, but it is enough to remove his mittens and shoes:
$l = [?_1, jacket, scarf, hat, ?_5, ?_6]$
and he can still keep the proofs of
$Mem(jacket(l))$, $Mem(scarf, l)$, $Mem(hat, l)$ and $Before(jacket, scarf, l)$
since they do not depend on the choices of $?_1, ?_5$ and $?_6$. The only thing left to do is to fill in l as
$l = [socks, jacket, scarf, hat, shoes, mittens]$
and to prove the remaining few properties.

3 Terms and Types

Terms in ALF are lambda terms, extended with constants and explicit substitution. Open terms, which contain free variables, must be validated in a *context*. A context is a sequence of typings of distinct free variables. A substitution is a sequence of simultaneous assignments of terms to variables

$$\{x_1 := a_1, x_2 := a_2, \ldots, x_n := a_n\}.$$

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 and a type α is

$$\alpha ::= \ \alpha_{ground} \mid (x_1 : \alpha_1; \ldots; x_n : \alpha_n)\alpha_{ground} \qquad \text{for } n > 0,$$

The syntax of terms is the following

$$e ::= \ x \mid c \mid [x_1, \ldots, x_n]e \mid e(e_1, \ldots, e_n) \mid c\{x_1 := e; \ldots; x_n := e\} \qquad \text{for } n > 0,$$

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 nor an abstraction and 3) in an explicit substitution $\{x_1 := e; \ldots; x_n := e\}$ applied to the constant c, the variables $x_1, \ldots, x_n$ must all be declared in the local context of c.

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 $\gamma\{x := a\}$).

4 Why is local undo difficult?

The complication in performing local undo properly is that there is information hidden in the visible output, which must be taken into account. Since we want local undo to be the dual operation of refinement, there are mainly two problems we must consider; instantiations caused by constraints and implicit sharing of subterms. We also need to make sure that the automatic instantiations and the constraints correspond to the proofterm after the undo operation, which means that the restrictions forced by the removed subterm should be removed, but all other restrictions kept.

Instantiations caused by constraints, which means that an unknown is instantiated because of a choice of *another* unknown, and if the latter is removed we expect the instantiation of the former to be removed as well, since it is not among the user refinements which should fully determine the resulting state. The instantiation is possible since the order in which unknowns are solved can be chosen freely, and a given solution to one unknown may force a particular instantiation to another unknown (due to dependent function types). For example, if mem is of type $(A : Set; a : A)Set$ and an unknown of type Set is refined (partially solved) with mem, the refined expression will become

$mem(?A, ?a) : Set$

and two new unknowns are created

$?A : Set$, and

$?a : ?A$.

The expression may be completed by either

(1) first solving $?A$ with N, and then $?a$ with 0, or

(2) directly solve $?a$ with 0

yeilding in both cases the solved expression

$mem(N, 0) : Set$

If the unknown $?a$ is solved directly, then $?A$ must be instantiated to N to make the term $mem(?A, 0)$ type correct, which means that the instantiation of $?A$ is a consequence of the choice of $?a$. Therefore, if the choice of $?a$ is withdrawn, the instantiation of $?A$ ought to be removed as well. On the other hand, if the user had first chosen $?A$ to be N, then the instantiation should not be effected since the choice of refining $?A$ with N is still a valid choice. Therefore, there will be a distinction between refinements by the user and instantiations forced by type checking. Note also that by refining $?a$ first, only one refinement was needed to complete the term, and this is what we meant by taking advantage of the type checking.

Implicit sharing of subterms Consider the following example: Suppose we want to prove that 2 divides 6, by solving the goal

$?x : DIV(2, 6)$

where $DIV(m, n) \equiv \exists k.m * k =_N n$ and $a =_N b$ is an abbreviation of $Id(N, a, b)$ which is the identity set with only one constructor *refl* stating reflexivity. The solution is obvious by choosing the witness k to 3, yeilding the term

$< 3, ?b >$ where $?b$ is of type $2 * 3 =_N 6$

and $?b$ is solved by reflexivity, yielding the proofterm $< 3, \mathit{refl}\ (2 * 3) >$. If we remove the choice of the witness k, we resume at

$<?k, \mathit{refl}\ (2{*}?k) >: DIV(2,6)$

where $?k$ is a new unknown with the restriction

$?k{+}?k = 6.$ $(2{*}?k \quad unfolded)$

Note that the two occurences of 3 was implicitly shared in $< 3, \mathit{refl}\ (2 * 3) >$, and the reason for this is that the first step of refinement we actually did was to refine the goal $?x : DIV(2,6)$ by the pairing operation, yeilding the term

$<?k, ?b >: DIV(2,6),$ where

$?k : N,$ and

$?b : 2{*}?k =_N 6$

The implicit sharing of $?k$ from this first step is remembered, and this is necessary to prevent illtyped terms ($<?k, \mathit{refl}\ (2{*}?3) >$ is **not** well typed). The restriction $?k{+}?k = 6$ comes from typechecking $\mathit{refl}\ (2{*}?k) : 2{*}?k =_N 6$.

Note that the state (proofterm) after removing the witness is a completely new state (proofterm).

If we now try to generalize the statement by removing the choice of $n = 6$, then the result becomes

$<?k, \mathit{refl}\ (2{*}?k) >: DIV(2, ?n)$

with the constraint

$?k{+}?k = ?n.$

Since the constraint uniquely determines $?n$, it is instantiated, yielding

$<?k, \mathit{refl}\ (2{*}?k) >: DIV(2, ?k{+}?k)$

If we now refine $?k$ with a variable and then abstract on that variable, we have generalised the proof to work for even numbers, which can be seen as a toy example of proof generalisation.

In the local undo procedure, we want to remove the minimum amount of instantiations to save useful work for the user. But we need to be careful not to loose necessary restrictions of the unknowns involved in the operation. Since several constraints may restrict the same unknowns, they may also interact in the process of searching automatic instantiations in a complicated manner. To avoid retracing the interaction between constraints and in favor of a faster refinement procedure, we have chosen an algorithm which recomputes the constraints after local undo. This will be discussed more in detail in section 9.

5 The scratch area

The scratch area contains two parts - a set of (non recursive) constant declarations $\mathcal{D}$ and a set of global constraints. The constraints are separated in two parts as well, the *simple* (or solved) constraints $\mathcal{C}_\mathcal{S}$ (corresponding to automatic instantiations) and the other constraints $\mathcal{C}$. A scratch area is denoted by $< \mathcal{D}, < \mathcal{C}_\mathcal{S}, \mathcal{C} >>$.

Declarations

A constant declaration is either *unknown*

$$u =? : \alpha \quad \Gamma$$

or defined by a flat term t_{flat}

$$u = t_{flat} : \alpha \quad \Gamma$$

where $t_{flat} ::= c(u_1, \ldots, u_n) \mid c\gamma(u_1, \ldots, u_n) \mid x(u_1, \ldots, u_n) \mid [x_1, \ldots, x_n]u$ for $(n \geq 0)$

c is a constant (from the environment or the scratch area),

γ is a substitution containing only flat terms,

x is a variable occurring in Γ,

and u is a constant in the scratch area.

For example,

$add = [x, y]u_1 : (x, y : N)N \quad [\,]$

$u_1 = natrec(u_2, u_3, u_4, u_5) : N \quad [x, y : N]$

$u_2 = [n]u_6 : (n : N)Set \quad [x, y : N]$

$u_6 = N : Set \quad [x, y, n : N]$

$u_3 =? : u_2(0) \quad [x, y : N]$

$u_4 =? : (w : N; v : u_2(w))u_2(succ(w)) \quad [x, y : N]$

$u_5 = y : N \quad [x, y : N]$

is a set of declarations. We will distinguish between *visible* and *invisible* constants, and we will use the convention of properly naming visible constants, whereas invisible constants will be denoted $u_1, u_2, \ldots$. The definitions of invisible constants will be expanded before presenting the scratch area for the user, yielding

$$add = [x, y]natrec([n]N, ?u_3, ?u_4, y) : (x, y : N)N \quad [x, y : N]$$

for the example above. The restriction of definitions to flat terms, means that we will have a (invisible) name for each proper subterm of any visible definition. Since the name is used in the other declarations, the problem of implicit sharing of subterms is taken care of.

Constraints

A constraint is an equation

$$e_1 = e_2 : \alpha \quad \Gamma$$

where at least one of e_1 and e_2 is an *incomplete* term. Constraints are denoted by quadruples $< e_1, e_2, \alpha, \Gamma >$.

Def An *incomplete* term is a term of the form

u	where u is an unknown constant
$u\gamma$	where u is an unknown constant and γ a substitution
$u(e_1, \ldots, e_n)$	where u is an unknown constant
$u\gamma(e_1, \ldots, e_n)$	where u is an unknown constant and γ a substitution
$b(e_1, \ldots, e_i, \ldots, e_n)$	where b is defined by pattern matching, i is the position

of a main argument of the function b, and e_i is an incomplete term.

In the last case, the incompleteness of a main argument e_i prohibits a match of any pattern defining the function b, since e_i is not on constructor form. The reason is that the list of patterns are guaranteed to be exhaustive and nonoverlapping ([Coq92]) which means that each position in the patterns are either all constructors or all variables.

Def A *simple constraint* is a constraint $< e_1, e_2, \alpha, \Gamma >$ where e_1 (or e_2) is an unknown constant $u \in \mathcal{D}$, and where e_2 (e_1 respectively) does not depend on u, i.e., the declaration is not circular.

The reason for requiring non circularity in a simple constraint, is that circular constraints (or cyclic sets of constraints) may actually appear in the global set of constraints. Since the simple constraints corresponds to the automatic instantiations of unknowns, the requirement is needed to guarantee non recursive declarations in $\mathcal{D}$. (This can be compared with the occurence check in unification algorithms). On the other hand, circularities in constraints may disappear as in the equation

$$u_i = u_j(u_i) : \alpha \quad \Gamma$$

which is a possible constraint. With u_j instantiated to a constant function $[x]c$, the equation reduces to the simple constraint

$$(u_i, c, \alpha, \Gamma).$$

The division of the scratch area in the parts $< \mathcal{D}, < \mathcal{C}_\mathcal{S}, \mathcal{C} >>$, gives us the possibility of distinguishing user refinements (the declarations with definitions in $\mathcal{D}$) from the unknowns which are forced to a unique solution relative to $\mathcal{D}$ (the simple constraints $\mathcal{C}_\mathcal{S}$). When the scratch area is presented, simple constraints are considered as ordinary refinements and all definitions of invisible constants are expanded.

6 Producing constraints

The set of global constraints in the scratch area originate in type checking the declarations in $\mathcal{D}$. The procedure is as follows. When an unknown constant is refined with a term e, e is checked to be of proper type, yielding a set of type equations that must be fulfilled for e to be type correct. The type equations are simplified to a set of term equations if the two types in each equation match structurally, and a type checking *failure* otherwise. The term equations are either reduced to a set of constraints by the conversion algorithm described below, or produce a *failure*. If the procedure was executed successfully so far, the set of constraints from the last step is added to $\mathcal{C}$ in the global constraints. Finally, the global constraints are simplified as far as possible and the result is checked to satisfies the requirements of an *admissible* refinement (described in section 8).

We will start by presenting an alternative type checking algorithm from the one given in [Mag92], which is independent of the reduction and conversion algorithms. Thereafter we will briefly present the conversion algorithm (= simplification of constraints), before we continue with the simplification of a global set of constraints.

6.1 Type checking

The term e has type α in context Γ in a given environment Σ, if the set of equations produced by the type checking algorithm has a solution, i.e there are instantiations of all unknowns occurring in e, α and Γ. If e, α and Γ are all complete, i.e., they contain no unknowns at all, then the algorithm becomes a decision procedure for type correctness since the only possible answers are an empty set of constraints or a failure. The type checking algorithm computes (denoted $\Longrightarrow$) a set of type equations (ξ), which contains tuples $< \alpha, \alpha', \Gamma >$ meaning that α and α' must be convertible in the context Γ. It proceeds by case analysis of the term structure.

$$\textbf{Var :} \quad \frac{x : \alpha' \in \Gamma}{x : \alpha \quad \Gamma \Longrightarrow \{< \alpha, \alpha', \Gamma >\}} \qquad \textbf{Const :} \quad \frac{c : \alpha' \in \Sigma}{c : \alpha \quad \Gamma \Longrightarrow \{< \alpha, \alpha', \Gamma >\}}$$

$$\textbf{Abs :} \quad \frac{e : \alpha \quad \Gamma + [x_1 : \alpha_1, \ldots, x_n : \alpha_n] \Longrightarrow \xi}{[x_1, \ldots, x_n]e : (x_1 : \alpha_1; \ldots, x_n : \alpha_n)\alpha \Longrightarrow \xi}$$

$$\textbf{App :} \quad \frac{\begin{array}{c} a_1 : \alpha_1 \quad \Gamma \Longrightarrow \xi_1 \\ a_2 : \alpha_2\{x_1 := a_1\} \quad \Gamma \Longrightarrow \xi_2 \\ \vdots \\ a_n : \alpha_n\{x_1 := a_1; \ldots; x_{n-1} := a_{n-1}\} \quad \Gamma \Longrightarrow \xi_n \end{array}}{f(a_1, \ldots, a_n) : \beta \quad \Gamma \Longrightarrow \bigcup_{i=1}^{n} \xi_i \cup \{< \beta, \beta'\gamma, \Gamma >\}}$$

where γ is $\{x_1 := a_1; \ldots; x_n := a_n\}$ and f has type $(x_1 : \alpha_1; \ldots; x_n : \alpha_n)\beta'$. Note that f must be a constant or a variable and can be looked up in the environment or context, respectively.

The transformation from type equations to term equations is straight forward and therefore omitted here. The simplification of term equations, also referred to as conversion, is defined in terms of reduction to head normal form. The reduction rules in question are β reduction (formulated in terms of explicit substitution), η expansion, and the ordinary λ calculus structural rules. Moreover, there are rules for explicit substitutions and expansion of simple constant definitions as well as pattern matching. We will leave out the actual rules, simply state that the result of the reduction is one of the following cases

- 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, which means that the term is not on head normal form neither is the term incomplete but it could not be reduced any further. The only possible situation is when the term is a function defined by pattern matching applied to all its arguments, where there is a variable (or an irreducible term) in the position of a main argument of the function.

- the term is incomplete, as defined above.

6.2 Conversion

The check for conversion is done in four stages. The first stage checks trivial cases such as syntactic equality or if either of the terms are unknowns. The second stage assures that the type of the terms are ground by transforming a conversion problem of higher type to an equivalent conversion problem of ground type. This is specially helpful in the pattern matching, since all functions will be applied to the proper number of arguments. The following stage reduces both terms to head normal form, irreducible form or incomplete form, respectively. The final stage investigate the form of the terms and invokes the (simpler) head conversion check when needed. The computation of the conversion problem, denoted by

$$Conv(e_1, e_2, \alpha, \Gamma)$$

results in a set of constraints (ξ), or a failure denoted by $Fail$, as shown below

Stage 1

If $e_1 \equiv e_2$ then $Conv(e_1, e_2, \alpha, \Gamma) \Longrightarrow \emptyset$ (empty set of constraints)

If either e_1 or e_2 is an unknown constant, then $Conv(e_1, e_2, \alpha, \Gamma) \Longrightarrow \{< e_1, e_2, \alpha, \Gamma >\}$

Stage 2

If α is a function type of arity n (i.e., $(x_1 : \alpha_1; \ldots ; x_n : \alpha_n)\beta$), then both e_1 and e_2 are applied to n *new* variables, say $y_1, \ldots, y_n$. The transformation corresponds to η-expansion followed by possibly α-conversion and removal of common abstraction. The transformed conversion problem will become

$$Conv(e_1(y_1, \ldots, y_n), e_2(y_1, \ldots, y_n), \beta\gamma, \Gamma')$$

where γ is the substitution $\{x_1 := y_1, \ldots, x_n := y_n\}$ and Γ' is the extension of Γ by the new variables $y_1, \ldots, y_n$.

Stage 3

Apply the reduction to head normal form to both terms, and depending on the status of the reduced terms e_1' and e_2', we take the following actions

- If any of the reduced terms are an incomplete term, then $Conv(e_1', e_2', \alpha, \Gamma) \Longrightarrow \{< e_1', e_2', \alpha, \Gamma >\}$
- If both terms are irreducible, then the head conversion algorithm is invoked.
- If both terms are on head normal form, then the head conversion algorithm is invoked.
- If neither of the above holds, (i.e., one term is on head normal form and one is irreducible), then the conversion fails, since the head of the irreducible term must be a function defined by pattern matching, whereas the head of the other is a constructor or a variable.

Stage 4

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 or irreducible, and since the type is ground, they are not abstractions. The only possibilities are for the heads to be variables, constructors or functions constants, 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

$$\frac{\begin{array}{c} Conv(a_1, a'_1, \alpha_1, \Gamma) \Longrightarrow \xi_1 \\ \vdots \\ Conv(a_n, a'_n, \alpha_n, \Gamma) \Longrightarrow \xi_n \end{array}}{Conv(b(a_1, \ldots, a_n), b'(a'_1, \ldots, a'_n), \beta, \Gamma) \Longrightarrow \bigcup_{i=1}^{n} \xi_i}$$

  else $Fail$ (for $b \not\equiv b'$).

* b **and** b'

$$Conv(b, b', \alpha, \Gamma) \Longrightarrow \begin{cases} \emptyset & \text{if } b \equiv b' \\ Fail & \text{otherwise} \end{cases}$$

7 Simplifying the global set of constraints

The simplification proceeds by applying the following rule of transformation, divided in two steps, to the global set of constraints $< \mathcal{C}_\mathcal{S}, \mathcal{C} >$:

1. Pick out a simple constraint $< u, e, \alpha, \Gamma >$ in $\mathcal{C}$ (if there are any), replace u by e everywhere in $\mathcal{C}_\mathcal{S}$ and $\mathcal{C}$ and add the simple constraint to $\mathcal{C}_\mathcal{S}$.

2. For each constraint in $\mathcal{C}$, replace the constraint with its corresponding set of (simpler) constraints, i.e., replace $< e_1, e_2, \alpha, \Gamma >$ by $Conv(e_1, e_2, \alpha, \Gamma)$ or fail if the conversion fails. This replacement only effects constraints which are not fully simplified, which means that the effected constraints previously contained the unknown u from above, which has been replaced by the term e.

This transformation will be performed until there are no more simple constraints to pick. It will terminate since
1) the number of unknowns are fixed, and for each transformation the number of unknowns occurring in $\mathcal{C}$ decreases by one, and
2) when a constraint is replaced by the simplification of the constraint, the result is either empty, identical to the original constraint or replaced with a set of constraints corresponding to the arguments compared pairwise which are all strictly smaller in complexity then the original constraint.

Def A *normal set of constraints* is a set $< \mathcal{C}_\mathcal{S}, \mathcal{C} >$ such that the rule of transformation is not applicable to the set.

Note that $< \mathcal{C}_\mathcal{S}, \emptyset >$ is trivially a normal set.

Def A set of simple constraints $\mathcal{C}_\mathcal{S}$ is an *independent substitution* if for any constraint $(u, e, \alpha, \Gamma) \in \mathcal{C}_\mathcal{S}$, u does not occur in any other constraints in $\mathcal{C}_\mathcal{S}$.

Proposition
Let $< \mathcal{C}_\mathcal{S}, \mathcal{C} >$ be a normal set of constraints obtained by the transformation rule above.

Then $\mathcal{C_S}$ is an independent substitution.

Proof: By induction on the number of simple constraints in $\mathcal{C_S}$. One simple constraint is an independent substitution by the definition of simple constraint and the only way to increase $\mathcal{C_S}$ is by the transformation rule first step, which preserves the property of independent substitution since the definition of the added constraint is expanded everywhere.

Conjecture 1.

Let $< \mathcal{C_S}, \emptyset >$ be a set of constraints. Then for each constraint $< u, e, \alpha, \Gamma >$ in $\mathcal{C_S}$, $e : \alpha \;\; \Gamma$ is a valid judgement for any instantiations of the unknowns occurring in the constraint.

The intuition is that since $\mathcal{C}$ is empty, there are no restrictions at all on the unknowns in the constraint, which means that we may assume the unknowns.

The following conjecture captures the idea of the state being determined by the partial proofterm:

Conjecture 2.

Let $< \mathcal{D}, < \mathcal{C_S}, \emptyset >>$ be a scratch area. Then $\mathcal{C_S}$ is uniquely determined from $\mathcal{D}$.

The intuition behind this conjecture is that the constraints in $\mathcal{C_S}$ are all simple constraints, i.e., there is only one possible instantiation for each unknown. This means that we have a solution to the unification problem (since $\mathcal{C}$ is empty there are no unsolved constraints left). What we claim is that this solution is unique. On the other hand, the conjecture is not true if $\mathcal{C}$ is not empty, unless we restrict ourselves to a selection algorithm which chooses the "smallest" unknown (in some ordering of unknowns), when there is a choice. The reason is that, for example, the set with the two constraints

$< u_1, f(u_2), \alpha, \Gamma >$ and $< u_2, g(u_1), \beta, \Delta >$

is either simplified to

$< u_1, f(u_2), \alpha, \Gamma >$ and $< u_2, g(f(u_2)), \beta, \Delta >$

or to

$< u_2, g(u_1), \beta, \Delta >$ and $< u_1, f(g(u_1)), \alpha, \Gamma >$

where in both cases the former constraint is simple and the latter circular (and therefore not simple). The result depends on which constraint is first chosen. However, we believe that this situation is only possible when the starting set of constraints is circular, and the circle of dependent constraints will eventually result in at least one cyclic constraint which is not simple by definition. Therefore, $\mathcal{C}$ can not become empty without first eliminating the circularity in the set of constraints.

8 Refinement

A refinement of an unknown constant u, where $u =? : \alpha \;\; \Gamma \in \mathcal{D}$ and u is not given a solution in $\mathcal{C_S}$, is to define u to be (the flat term) e. The refinement procedure is done in the following five steps

1. e is checked to be of type α in context Γ, producing a set of type equations.

2. The set of equations is simplified, producing a set of constraints ξ or a failure.

3. If 2) succeeded, ξ is added to the global set of constraints and the entire set is simplified, i.e.,

$$<\mathcal{C}_{\mathcal{S}}, \mathcal{C} \cup \xi> \longrightarrow <\mathcal{C}'_{\mathcal{S}} \cup \mathcal{C}_{\mathcal{S}u}, \mathcal{C}'> \text{ or } \mathit{FAIL}$$

   where $\mathcal{C}_{\mathcal{S}u}$ is the new simple constraints which are consequences of the refinement of u.

4. If 3) succeeded, we must check that the solutions given in $\mathcal{C}_{\mathcal{S}u}$ all fit their scopes, i.e., if
   $<u, e, \alpha, \Delta> \quad \in \mathcal{C}_{\mathcal{S}u}$
   and
   $u = ? : \alpha \quad \Gamma \quad \in \mathcal{D}$
   then we must check that all free variables in e is in the context Γ. An example of this problem is given below.

5. If 4) succeeded, then the refinement is *admissible*, and results in the new scratch area

$$<\mathcal{D}, <\mathcal{C}_{\mathcal{S}}, \mathcal{C}>> \longrightarrow <\mathcal{D}', <\mathcal{C}'_{\mathcal{S}} \cup \mathcal{C}_{\mathcal{S}u}, \mathcal{C}'>>$$

   where $\mathcal{D}' = \mathcal{D}$ with ? replaced by e in the declaration of u.

The problem of a solution in $\mathcal{C}_{\mathcal{S}}$ not being in the proper scope can only arise when an unknown u with context Γ is *used* in the definition of another declaration u' (with context Δ) and $\Gamma \subset \Delta$, i.e., Γ is a proper subcontext of Δ. (The converse is not possible since the variables in Γ are "free in u", and must be declared in the context where u is used). Moreover, u must get its instantiation because of the way it is used in u', i.e., u will occur in a constraint produced from type checking the declaration u', which has access to the larger context Δ. We will illustrate the scoping problem with the example of trying to prove the proposition $\exists x.\forall y.R(x, y)$ for some relation R.

Ex. Suppose there is a constant
$\mathit{refl} : (x : a)R(x, x) \quad [\,]$
and the scratch area declarations
$\mathit{proof} = ? : R(?x, y) \quad [y : A]$
$x = ? : A \quad [\,]$
which come from the proposition above. Refining *proof* with the constant *refl*, creates a new unknown constant u (for the argument of *refl*) and yields the declarations
$\mathit{proof} = \mathit{refl}\ (?u) : R(?x, y) \quad [y : A]$
$x = ? : A \quad [\,]$
$u = ? : A \quad [y : A]$
The type of *refl* $(?u)$ is $R(?u, ?u)$, so the result of type checking the declaration of *proof* is the equation $(R(?u, ?u) = R(?x, y) : \mathit{Set} \quad [y : A])$ which is simplified to
$u = ?x : A \quad [y : A]$
$u = y : A \quad [y : A]$
which results in the two simple constraints
$u = y : A \quad [y : A]$
$x = y : A \quad [y : A]$
but where $?x$ is instantiated to y, which is **not** in the scope of $?x$ (since the local context of x does not contain y).

Conjecture 3. If u is given a solution in $\mathcal{C}_{\mathcal{S}}$ which is not in the scope of u, then u has no solution.

The argument is that if u is given a solution in $\mathcal{C}_{\mathcal{S}}$, then that is the only possible instantiation (by unification) and since it is not a valid solution, there is none.

8.1 User friendly refinement

We have described the atomic refinement, when the term e to refine with is restricted to a flat term. Since the first step in the atomic refinement is to type check e, all constants in e must be known before the refinement. Neither of these two restrictions are particularly user friendly, and therefore the system preprocesses the user refinement and generates the new unknown constant declarations before the actual refinement. Moreover, any user refinement $u = e$ where e is not flat, can easily be transformed into a sequence of refinements (with flat terms) by investigating the term structure.

Recall that a flat term is either an abstraction $[x_1, \ldots, x_n]u$ or $b(u_1, \ldots, u_n)$ where b is either a constant or a variable. It is the u in the abstraction and the arguments $u_1, \ldots, u_n$ in the application we want the system to generate new unknown declarations for. Since in the abstraction there is nothing the user must contribute with, there is a special command that given an unknown constant u

$$u =?: (x_1 : \alpha_1; \ldots; x_n : \alpha_n)\alpha \quad \Gamma$$

performs the abstraction by adding a new constant u'

$$u' =?: \alpha \quad \Gamma + [x_1 : \alpha_1; \ldots; x_n : \alpha_n]$$

and invokes the atomic refinement

$$u = [x_1, \ldots, x_n]u'$$

In the application case, the user supplies the constant or variable, and its type is looked up and compared to the type of the constant to be refined. If the constant/variable needs to be applied to, say k, arguments to make the arities of the types match, k new declarations are added:

$$\begin{array}{l} u_1 =?: \alpha_1 \quad \Gamma \\ u_2 =?: \alpha_2\{x_1 := u_1\} \quad \Gamma \\ \vdots \\ u_k =?: \alpha_k\{x_1 := u_1 \ldots x_{k-1} := u_{k-1}\} \quad \Gamma \end{array}$$

if $b : (x_1 : \alpha_1; \ldots; x_n : \alpha_n)\beta \quad \Delta$, where $k \leq n$, and the unknown u to be refined is in context Γ, where $\Delta \subset \Gamma$. Finally, the atomic refinement

$$u = b(u_1, \ldots, u_k)$$

is invoked.

9 Local undo

As mentioned, by local undo we mean "withdraw an earlier made choice including its consequences". This can now be described in our setup by replacing a declaration in $\mathcal{D}$

$u = e : \alpha \;\; \Gamma$

by

$u = ? : \alpha \;\; \Gamma$

which withdraws the choice. To remove the consequences, we need to undo the effect the set of constraints produced from type checking $e : \alpha \;\; \Gamma$, had on the global set of constraints. This could be done in two different ways. One way is to keep the original constraints in the global set of constraints, and perform the simplification each time it is needed. Then the constraints produced from type checking the declaration u could simply be removed. Since the simplification involves reduction of terms as well as expansion of simple constraints, it is obviously an irrevocable procedure. The other alternative is therefore to type check each declaration in $\mathcal{D}$ again, to recover the original constraints. We have chosen the latter alternative, since local undo is normally performed very seldom compared to refinements, and to simplify the set of constraints after each refinement would be time consuming, but necessary to present the scratch area in a satisfactory way.

Making the declaration of u into an unknown is a necessary condition, but not sufficient to achieve a useful algorithm. It would be like a functional language implementation without a garbage collector, since making u unknown removes the access to all the invisible constants corresponding to subterms of e, and the scratch area would soon be cluttered with useless declarations. Therefore, we must possibly also delete entire declarations in $\mathcal{D}$ when removing the term e.

It might be helpful to picture the declarations as graphs, where nodes correspond to constants declared in the scratch area, and an edge from a to b means that b is used in the definition of a. All visible constants can be seen as root nodes of graphs and the invisible constants as nodes of subgraphs. The graphs may be connected if a visible constant is used in another constant declaration. With this intuition in mind, the declarations that should be deleted are the constants corresponding to nodes in the term graph which can be reach from the node u without visiting the root node of a visible constant. Since visible constants can be accessed by the user, it may be used in other places than in e and should not be deleted just because it is used in a deleted term. We will distinguish between explicit sharing which is when the user deliberately uses the same (visible) constant in several places, and the implicit sharing which is introduced when an unknown is refined with a constant or variable with actually dependent type. Then we have the following properties

- Implicit sharing only occurs within a visible constants subgraph.
- Explicit sharing corresponds to an edge from within a subgraph to the root of another subgraph.
- Only the root of a subgraph could be reached from other subgraphs.
- The declarations to be garbage collected, are the constants in the subgraph of the deleted constant, and any explicit sharing edge within that subgraph is removed.

More formally, removing the definition of declaration u (which can correspond to any proper subterm in the visible part of the scratch area) is to transform the scratch area in two steps. First the required declarations in $\mathcal{D}$, and the set of constraints are removed, and then the recomputation is done.

$$< \mathcal{D}, < \mathcal{C}_{\mathcal{S}}, \mathcal{C} >> \longrightarrow < \mathcal{D}', < \emptyset, \emptyset >> \longrightarrow < \mathcal{D}', < \mathcal{C}'_{\mathcal{S}}, \mathcal{C}' >>$$

with $\mathcal{D}' = \mathcal{D} - \mathcal{D}_u$ where $\mathcal{D}_u$ is the set of declarations corresponding to $u's$ subgraph and where the declaration of u is made unknown,
and $< \mathcal{C}'_{\mathcal{S}}, \mathcal{C}' >$ is the set of constraints produced from type checking each declaration in $\mathcal{D}'$ yielding a new global set of constraints, which is simplified as far as possible.

Conjecture 4.

Let $\mathcal{C}_{\mathcal{S}}$ be a set of simple constraints and $\mathcal{C}'_{\mathcal{S}}$ the result after performing a local undo operation. If the simplification of global constraints is performed with a deterministic selection algorithm, then $\mathcal{C}'_{\mathcal{S}} \preceq \mathcal{C}_{\mathcal{S}}$, where $\preceq$ means that

1) the constants instantiated in $\mathcal{C}'_{\mathcal{S}} \subseteq$ the constants instantiated in $\mathcal{C}_{\mathcal{S}}$, and

2) if u has definition e' in $\mathcal{C}'_{\mathcal{S}}$ and definition e in $\mathcal{C}_{\mathcal{S}}$, then there is an instantiation of (some of) the unknowns in e', such that the two definitions become identical.

The intuition behind this conjecture is that the state of the scratch area after a local undo, could have been reached without the undo operation but with the refinements done in another order. Moreover, if the constants of the simple constraints in $\mathcal{C}'_{\mathcal{S}}$ are uniquely determined from the new declarations, then additional refinements could not change the already determined instantiations, and $\mathcal{C}'_{\mathcal{S}}$ would be less defined in this particular meaning than $\mathcal{C}_{\mathcal{S}}$.

10 Conclusion: How are the problems in the introductory section solved?

The problem with implicit sharing is solved since there is an internal name for each proper subterm (an invisible constant declaration), and internally the name is used instead of the term at all its shared occurrences. Therefore, when removing a subterm, the corresponding invisible constant declaration is made into an unknown constant, which means that all *shared* occurrences of that subterm will become the (same) unknown constant (i.e., a new goal).

The two other problems; removing "consequences of a choice", i.e., undo the effect that a particular choice had on the scratch area, and the problem of achieving restrictions on the new unknowns (which are the least possible restrictions), are both related to how instantiations by constraints are treated. Since internally we distinguish between unknowns refined by the user and unknowns instantiated by constraints, these problems can be solved. When a subterm is removed, the corresponding internal constant declaration is made unknown and unreachable declarations are garbage collected. This means that the set of declarations exactly corresponds to the choices made by the user *except* the one just withdrawn. Therefore, the resulting set of constraints yielded by type checking the remaining declarations, will only contain restrictions caused by other refinements and they will be as unrestrictive as possible.

11 Optimisations in the implementation

Since ALF is an interactive proof editor equipped with an window interface, the presentation of the scratch area to the user is of great importance. As already mentioned, invisible constants and solved constraints are expanded before the scratch area is presented to the user. Naturally, changes in the scratch area ought to be apparent after each action by the user. It would be rather time consuming to perform all these expansions each time an action is taken which alters the scratch area. Therefore, we have chosen to include a *visible part* of the scratch area, which is exactly what is presented to the user. This means that some information is represented twice, in favour of speed. The only time expansions of invisible constants and simple constraints need to be performed with this optimisation, is after local undo.

There is also some minor optimisations, such as keeping a *dependency graph* reflecting the dependency between the constants in the scratch area and by giving invisible constants names corresponding to their path in a visible constant declaration. The naming convention is practical to get direct access to a declaration from its position in the visible part, as well as for "garbage collecting" declarations after local undo. The dependency graph is mainly used for circularity checks, which is required in every refinement. But it is also convenient when checking if a declaration is complete (when moved to the environment) or for sorting the declarations in dependency order (used for a quicker expansion of invisible constants as well as for printing on external files). There is another optimisation (which is not yet implemented but will be if needed) and that is to only include invisible declarations of which the constant does occur in several places. This means that only subexpressions which are actually shared implicitly are given a name, since the others need not be named for a proper local undo algorithm.

Acknowledgements
I would like to thank my supervisor Thierry Coquand and Bengt Nordström for many valuable discussions and suggestions concerning this work, Lennart Augustsson for commenting on this paper and Gustavo Betarte for commenting an earlier version of this paper.

References

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[Hue75] G.P. Huet. A unification algorithm for typed λ-calculus. In *Theoretical Computer Science, 1(1):27-57, 1975.*

[Mag92] Lena Magnusson. The new Implementation of ALF. In *the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[Mäe93] Petri Mäenpää. The Art of Analysis. Logic and History of Problem Solving In *Ph.D thesis, University of Helsinki, September 1993*

[TBK92] L. Thry, Y. Bertot, and G. Kahn. Real Theorem Provers deserve real user-interfaces. In *INRIA Technical Report no. 1684, May 1992*

Encoding Z Schemas in Type Theory

Savi Maharaj*
Savi.Maharaj@dcs.ed.ac.uk

15 September 1993

Abstract

This report describes methods of representing the Z Schema Calculus in the type theory UTT. We first attempt a direct encoding of schemas as Σ-types in the manner prescribed by Luo. This turns out to be unsatisfactory because encoding the operations of the Schema Calculus requires an ability to perform computations on the syntax of schemas, so we develop methods in which this syntax is also represented. These methods also depend upon the existence of Σ types but use them in an unconventional fashion.

1 Introduction

The Z language[BN92, Spi88] is an formal notation which provides an expressive, unambiguous language for writing specifications of programs. One of its main strengths is a module-handling mechanism called the Schema Calculus which allows specification modules to be put together in various ways to build new specifications. However Z falls short of providing a complete program development *methodology* in that it lacks a notion of implementation, and provides no mechanical support for carrying out proofs.[1].

The Unifying Theory of dependent Types [Luo91b], and its implementation in the LEGO proof-checker [LP92], possess features that complement Z to some degree. LEGO provides a means of doing machine-checked proofs in UTT, while UTT provides us with notions of "program" and "proof" which we can use to define an implementation. However UTT lacks some of the user-friendly properties of the specification languages used in industry. One goal of this work is to explore methods in which we can put together the tried and tested expressiveness of the Z notation with UTT and LEGO.

In this paper we investigate ways of combining Z and UTT by encoding a portion of the Z Schema Calculus into UTT. This enriches UTT by providing a way of writing structured specifications in UTT in the style of Z. This work can also be viewed as a presentation of a possible semantics for the Z Schema Calculus in UTT, giving us an opportunity to use LEGO to explore the properties of the Z Schema Calculus. We hope to gain insight into the meaning of Z schemas and to gather a collection of LEGO proofs about the properties of of Z schemas which we can then use for proving properties of specifications and for doing program verification.

For instance, Z provides an operation whereby two schemas S and T can be conjoined to produce a new schema $S \wedge T$. We would like to know whether if we have an implementation of schema $S \wedge T$ we can always derive from it implementations of S and of T — a kind of elimination

*University of Edinburgh. This work was partially supported by the British Council.

[1] There have been efforts to remedy some of these problems.

rule. Conversely, if a program implements both S and T, or is an extension of implementations of these, does it then yield an implementation of $S \wedge T$ — a kind of introduction rule?

In section 2 I introduce a small fragment of the Z notation, just enough to illustrate the work that I have done. Then in section 3 I discuss the ways in which I have tried to encode Z schemas in LEGO, showing the problems that have led me to develop each subsequent technique from the previous one. Finally in section 4 I talk about the theorems I've managed to prove and discuss some outstanding conjectures, questions, and problems.

2 The Z notation

The Z notation consists of a core language based on set theory, and a structuring mechanism consisting of modules called *schemas* which can be combined in various ways using operations that make up the *Schema Calculus*.

Here is an example of a simple Z schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\
\hline
x : \mathbb{N} \\
y : \mathit{list}\ \mathbb{N} \\
\hline
x \leq \mathit{length}\ y \\
\hline
\end{array}$$

The schema consists of a *signature* and a *predicate*.

One of the operations of the schema calculus is schema conjunction; this involves *join*ing the signatures of two schemas and conjoining their predicates to give a new predicate over the joined signature. When two signatures are joined occurrences of the same identifier in both signatures are identified with each other.

Example:

$$\begin{array}{|l}
\multicolumn{1}{l}{T} \\
\hline
x : \mathbb{N} \\
z : \mathit{list}\ \mathbb{N} \\
\hline
x \geq \mathit{length}\ z \\
\hline
\end{array}$$

Conjoining S and T gives the schema:

$$\begin{array}{|l}
\multicolumn{1}{l}{S \wedge T} \\
\hline
x : \mathbb{N} \\
y : \mathit{list}\ \mathbb{N} \\
z : \mathit{list}\ \mathbb{N} \\
\hline
(x \leq \mathit{length}\ y) \wedge (x \geq \mathit{length}\ z) \\
\hline
\end{array}$$

Other operations, which I won't describe here, include disjunction, implication, negation, inclusion of one schema within another, composition of schemas, and piping. Most of these make use of the *join* operation to combine signatures.

3 Representing Schemas in Type Theory

A note about the core language

The core language of Z is based on Zermelo-Fraenkel set theory. While it is possible to encode sets in UTT in various ways [Mah90, CM93], there is no reason to attempt this if we are mainly interested in the schema calculus since we can use UTT itself as our core language. If we find a flexible representation of schemas, we can later change our representation of the core language, if desired, without affecting the theorems we prove about the module language.

We can represent the types used by Z by UTT types, including a selection of inductive types such as *nat*, *bool* and polymorphic *list*. See [LP92] and [Luo91b, Luo93] for discussions of, respectively, the practical and the theoretical aspects of introducing such inductive datatypes into the LEGO implementation of UTT. Note that we will also be using these types as our metalanguage in which we encode Z schemas.

Method 1: Schemas as Σ types

We can use Σ-types to represent schemas in a manner similar to the use of type theory outlined in [Luo93]. Then the schema S presented above is encoded in UTT as:

$$
\begin{aligned}
&S\_sig \stackrel{\mathrm{def}}{=} nat \times list\ nat \\
&S\_pred \stackrel{\mathrm{def}}{=} \lambda str : S\_sig.\ str.1 \leq length\ str.2 \\
&S \stackrel{\mathrm{def}}{=} \Sigma\, str : S\_sig.\ S\_pred\ str
\end{aligned}
$$

This allows us to define an implementation: an implementation of S is just an object whose type is S, that is, a program of type S_sig paired with a proof that this satisfies the predicate S_pred. But how do we go about defining the operations of the schema calculus? In order to implement the *join* operation we need to compare the identifiers and types used in our schemas. But these things have no existence as objects within the type theory.

It may appear that our problem would be solved by adding record types to the type theory. In fact this is not so, since the problem is not the absence of labels for the components of the Σ-type, but the impossibility of treating such labels as terms in the type theory that can be acted upon by functions defined in the type theory. What we need to do therefore is to encode the *syntax* of Z identifiers and types as terms in UTT that we can use to do computations.

Method 2: Syntactic names and types in signatures

We introduce two new inductive types to represent identifiers and type names. The identifiers will be derived from the specification that is being encoded. For our example we have:

$$Ident\ ::=\ \text{`}x\text{'} \mid \text{`}y\text{'} \mid \text{`}z\text{'}$$

We also introduce a new inductive type of Z type names. Here, $Given\_Type$ is an inductive type consisting of type names over which a specification is parameterised.

$$Ztype\ ::=\ natT \mid boolT \mid givenT\ \ Given\_Type \mid funT\ \ (Ztype, Ztype) \mid prodT\ \ (Ztype, Ztype)$$

Now we define a signature as a list of pairs of these syntactic identifiers and types:

$$Signature \stackrel{\mathrm{def}}{=} list\ (Ident \times Ztype)$$

Since we still want to be able to relate these specifications to programs written in UTT, we must define the relationship between these syntactic signatures and types in UTT. To do this we define a semantic function, *Typ* that maps these syntactic types to UTT types: e.g. $Typ\ natT = nat$. Then we extend this to a function *Typify* of type $Signature \to Type$[2] which forms a product of the types obtained by applying *Typ* to all of the *Ztype*s in a given signature, together with the unit type in the case of the empty signature. This essentially allows us to recapture our previous definition of a signature as a product of types.

We can then define a schema as consisting of a syntactic signature *Sig* paired with a predicate over the semantic signature *Typify sig*:

$$Schema \stackrel{\text{def}}{=} \Sigma\, sig : Signature.\, (Typify\ sig) \to Prop$$

We define an implementation of such a schema $(sig, pred)$ as an program of type *Typify sig* paired with a proof that it satisfies the predicate *pred*. This closely resembles the notion of implementation in Method 1.

$$Imp \stackrel{\text{def}}{=} \lambda S : Schema.\, \Sigma\, str : Typify\ S\_sig.\, S\_pred\ str$$

In order to work with these syntactic functions we need to define several functions. One of these is *lookup* which, when given an identifier and a tuple *str* of type *Typify sig* for some signature *sig*, attempts to locate the identifier in *sig* and then returns the value in the corresponding position in *str*. We use sums to handle failure, returning $in1\ void$ if the given identifier does not appear in the given signature.

One of the uses of *lookup* is in writing schema predicates. This is illustrated by the following example of a schema:

$$\begin{aligned}
&S_sig \stackrel{\text{def}}{=} [(\text{`}x\text{'}, natT), (\text{`}y\text{'}, listT\ natT)]\\
&S_pred \stackrel{\text{def}}{=} \lambda str : Typify\ S_sig.\, (lookup\ \text{`}x\text{'}\ str) \le (lookup\ \text{`}y\text{'}\ str)\\
&S \stackrel{\text{def}}{=} (S_sig, S_pred)
\end{aligned}$$

Another example of a schema is the trivial, unsatisfiable, *Absurd_schema* which consists of the empty signature and the predicate $\lambda str : Typify\ [\,].\, absurd$.

Now we can begin to define the operations of the schema calculus. Consider the operation of conjoining two schemas S and S'. First we must form a new signature *newsig* by joining the signatures of S and S'. These two signatures may be inconsistent, in that there may be an identifier which occurs in both of them that is paired with different *Ztype*s in each occurrence, so we must find some way of handling this possibility. Next we must form a new predicate over the semantic signature *Typify newsig* by conjoining the two old predicates. To do this we have to make use of coercions which map programs in the new semantic signature back to programs in *Typify sig* and *Typify* sig' since these types are the domains of the old predicates.

First we define the *join* function. This takes two signatures *sig* and sig' yields a new signature *newsig* together with a coercion back from *newsig* to *sig*. The coercion takes a tuple of type *Typify newsig* and produces a tuple of type *Typify sig* by projecting only those components that correspond to identifiers present in *sig*. No coercion back to the second signature sig' is computed since this may not exist if *sig* and sig' happen to be inconsistent.

[2]Actually, $Type(0)$, but for simplicity we ignore details about type universes in this paper.

Another operation *coerce* attempts to find coercions between arbitrary signatures. The type of *coerce* reflects the fact that it is partial: if no coercion exists it returns $in2\ void$.

$$\Pi\, S, S' : Signature.\, ((Typify\ S) \to (Typify\ S')) + unit$$

Now we can define schema conjunction. To conjoin S and S' we first *join* their signatures to form a new signature *newsig*. Then we attempt to coerce *newsig* back to the signature of S'. This will fail if S and S' happen to be inconsistent, in which case we return $Absurd\_schema$ as our result. Otherwise we return a schema made up of *newsig* and the predicates of S and S' conjoined and composed with coercions as appropriate.

The definition is as follows. I use square brackets [.] here to enclose local definitions of identifiers, a useful feature of LEGO syntax.

$$\begin{array}{l}
And_schema \stackrel{\text{def}}{=} \lambda S, S' : Schema. \\
\quad [tmp \stackrel{\text{def}}{=} join\ S\ S'] \\
\quad [newsig \stackrel{\text{def}}{=} tmp.1] \\
\quad [coercion1 \stackrel{\text{def}}{=} tmp.2] \\
\quad [coercion2 \stackrel{\text{def}}{=} coerce\ newsig\ s'.1] \\
case \\
\quad (\lambda f : (Typify\ newsig) \to (Typify\ S'.1). \\
\qquad (newsig, \lambda s : Typify\ newsig.\, (S.2\ (coercion1\ s)) \wedge (S'.2\ (f\ s)))) \\
\quad (\lambda x : unit.\, Absurd_schema) \\
\quad coercion2
\end{array}$$

With this method we can define all the operations of the schema calculus. However we need to compute lots of coercions and reasoning about these turns out to be difficult. For instance, *coerce* makes use of a simple function called *coerce_Ztypes* which takes two *Ztype*s z and z' and attempts to find mappings to and from $Typ\ z$ and $Typ\ z'$. Its type is the following:

$$\Pi\, z, z' : Ztype.\, (((Typ\ z) \to (Typ\ z')) \times ((Typ\ z') \to (Typ\ z))) + unit$$

This function is defined by induction on *Ztype*s in a straightforward way. It succeeds when z and z' are identical, in which case it returns functions that are extensionally equal to the identity function on the appropriate type. So I tried to prove the following result:

$$\begin{array}{l}
\forall z : Ztype.\, \exists f, f' : (Typ\ z) \to (Typ\ z). \\
\quad (coerce_Ztypes\ z\ z = in1\ (f, f')) \wedge (\forall x : Typ\ z.\, (f\ x = x) \wedge (f'\ x = x))
\end{array}$$

However I found that in order to prove I needed to assume extensionality for functions. The reason for this has to do with way that *coerce_Ztype* is defined on *Ztype*s of the form $funT\ z\ z'$. The coercions that are found in this case are of type $((Typ\ z) \to (Typ\ z')) \to ((Typ\ z) \to (Typ\ z'))$ They map a function f to another function created by pre- and post-composing f with coercions between z and z and between z' and z'. This function is only extensionally equal to f.

Extensionality is not part of UTT and we would prefer not to have to assume it. Perhaps we can avoid it if we find a way of avoiding having to find coercions. We needed coercions because schema predicates were defined over specific signatures and therefore needed to be composed with coercions before they could be applied to other signatures. So what if we allow our predicates to be defined over all signatures? This leads us to explore a third method of representing Z schemas.

Method 3: Syntactic names in programs

The idea here is to introduce a new type of programs, over which schema predicates will be defined. Whereas before we used UTT's typing rules to determine whether a program matched a signature, we will now have to explicitly define a matching relation between programs and signatures.

We continue to use the types *Ident*, *Ztype* and *Signature* and the semantic function *Typ*. However instead of representing programs as tuples, we use a more syntactic representation in which values are associated with syntactic names and types. I call these programs *Structure*s since they are reminiscent of Structures in the programming language SML.

$$Structure \stackrel{\text{def}}{=} list\ (\Sigma\, p : Ident \times Ztype.\ Typ\ p.2)$$

We have to write some functions to handle these syntactic programs. The types of three of these are given below. The first function, *match*, checks whether the names and identifiers in a *Structure* are exactly the same as those in a given *Signature*. The second, *restrict*, attempts to cut down a structure so that it has only those components specified in a given *Signature*; *restrict* fails if the *Signature* requires components that are not in the *Structure*. We will use the third function, *lookup*, in writing schema predicates.

$$\begin{aligned}&match : Signature \to Structure \to bool\\ &restrict : Structure \to Signature \to unit + Structure\\ &lookup : \Pi\, p : Ident \times Ztype.\ Structure \to unit + (Typ\ p.2)\end{aligned}$$

An example of a syntactic program is the following:

$$prog \stackrel{\text{def}}{=} [(\text{`}x\text{'}, natT, 0), (\text{`}b\text{'}, boolT, true)] : Structure$$

If we define sig to be the signature $[(\text{`}b\text{'}, boolT)]$ then $(match\ sig\ prog)$ evaluates to *false* and $(restrict\ prog\ sig)$ evaluates to the *Structure* $[(\text{`}b\text{'}, boolT, true)]$.

Our new definition of the type *Schema* is:

$$Schema \stackrel{\text{def}}{=} Signature \times (Structure \to Prop)$$

Schema predicates become more complicated since we always have to allow for the possibility that *lookup* might fail. The predicates and relations that we use in writing schema predicates need to be redefined with this in mind. For instance in the definition of the schema S we replace $\leq : nat \to nat \to Prop$ with $\preceq : (unit + nat) \to (unit + nat) \to Prop$ which is defined as the proposition *absurd* in the case where either of its first arguments happens to be $in1\ void$. Here is the new definition of this schema:

$$\begin{aligned}&S\_sig \stackrel{\text{def}}{=} [(\text{`}x\text{'}, natT), (\text{`}y\text{'}, listT\ natT)]\\ &S\_pred \stackrel{\text{def}}{=} \lambda str : Structure.\ (lookup\ \text{`}x\text{'}\ natT\ str) \preceq length\ (lookup\ \text{`}y\text{'}\ (listT\ natT)\ str)\\ &S \stackrel{\text{def}}{=} (S\_sig, S\_pred)\end{aligned}$$

Another example is the new definition of *Absurd_Schema*, which now has as its predicate $\lambda str : Structure.\ absurd$.

We need to place a condition on schemas in order to exclude some badly behaved predicates. We would like predicates to remain true of structures if they are enriched by adding new components. Therefore we don't want to allow, for instance, a predicate that says that a certain identifier does not occur in a structure. The following condition excludes predicates of this sort.

$$\mathit{Valid\_schema} \stackrel{\text{def}}{=} \lambda S : \mathit{Schema}. \forall \mathit{str} : \mathit{Structure}. (S.2\ \mathit{str}) \Leftrightarrow (S.2\ (\mathit{restrict}\ \mathit{str}\ S.1))$$

The user will need to prove that this condition is satisfied whenever a new schema is defined. We will have to show that all our basic schemas and schema operations have or preserve this property. It is trivial to show that *Absurd_Schema* has this property.

With syntactic programs like these, our definitions of schema operations become much simpler since there is no longer any need to compute or keep track of coercions. The function *join*, for instance, gets redefined in a simpler way.

Here is the new definition of schema conjunction. We make use of a new function *consistent* : $\mathit{Signature} \rightarrow \mathit{Signature} \rightarrow \mathit{bool}$ which checks whether two signatures are consistent.

$$\begin{array}{l}
\mathit{And_schema} \stackrel{\text{def}}{=} \lambda S, S' : \mathit{Schema}. \\
\mathit{if} \\
\quad (\mathit{consistent}\ S.1\ S'.1) \\
\quad (\mathit{join}\ S.1\ S`.1,\ \lambda s : \mathit{Structure}. (S.2\ s) \wedge (S'.2\ s)) \\
\quad \mathit{Absurd_schema}
\end{array}$$

An implementation of a schema S is a structure *str* paired with a proof of (*Implements* S *str*) where *Implements* is defined as

$$\lambda S : \mathit{Schema}. \lambda \mathit{str} : \mathit{structure}. (\mathit{match}\ S.1\ \mathit{str} = \mathit{true}) \wedge (S.2\ \mathit{str})$$

This states that an implementation of a schema is a *Structure* which exactly matches the schema's signature and which satisfies the schema's predicate. It may turn out that requiring an exact match with the signature is too restrictive, but for now this is the definition that we use.

4 Results

So far I have had partial success in proving the kinds of results that I want about my encoding of the Z schema calculus. The theorems I have proved depend on a number of somewhat baroque but uncontentious conjectures about the behaviour of the functions *join* and *restrict*. I am hoping that my attempts to prove these conjectures will uncover cleaner and more illuminating properties that can take their place.

I won't list all these conjectures but only give a couple of examples. The first of these is typical. This says that a *Structure* that matches the *join* of two consistent *Signature*s can successfully be *restrict*ed to the first *Signature*. The corresponding property regarding the second *Signature* is also one of my conjectures.

Conjecture 1

$$\begin{array}{l}
\forall S, S' : \mathit{Signature}. \forall \mathit{str} : \mathit{Structure}. \\
\quad (\mathit{consistent}\ S\ S' = \mathit{true}) \Rightarrow \\
\quad (\mathit{matches}\ (\mathit{join}\ S\ S').1\ \mathit{str} = \mathit{true}) \Rightarrow \\
\quad \exists \mathit{str}' : \mathit{Structure}. \mathit{restrict}\ \mathit{str}\ S = \mathit{in2}\ \mathit{str}'
\end{array}$$

The next is the only conjecture that does not involve the *restrict* function. It states that that if there is some structure which satisfies the predicates of both of two valid schemas, then those schemas have consistent signatures.

Conjecture 2

$$\begin{array}{l}\forall S, S' : Signature . \forall str : Structure . \\ \quad (Valid\_schema\ S) \wedge (Valid\_schema\ S') \wedge (S.2\ str) \wedge (S'.2\ str) \Rightarrow \\ \quad consistent\ S.1\ S'.1 = true\end{array}$$

Under the assumption that these, and similar, conjectures are true I have been able to carry out some proofs in LEGO about schema conjunction. My first theorem states that schema conjunction preserves the property *Valid_schema*:

Proposition 1 (And_preserves_valid_schema)

$$\begin{array}{l}\forall S, S' : Schema . (Valid\_schema\ S) \wedge (Valid\_schemaS') \Rightarrow \\ \quad Valid\_schema\ (And\_schema\ S\ S')\end{array}$$

My proof of this takes slightly over 100 steps in LEGO and makes use of a number of conjectures about the behaviour of *restrict*.

The next three results make use of a predicate named *restricts_to_impl* which has type $Schema \rightarrow Structure \rightarrow Prop$. This says two things: that a given *Structure* can be successfully restricted to the signature of a given *Schema*, and that the new *Structure* obtained by this *restriction* is an implementation of the given *Schema*.

The following result gives a kind of introduction rule for schema conjunction. If we are given a *Structure* *str* which can be restricted to give implementations of two *Schemas* S and S', then we can conclude from this theorem that *str* can be restricted to an implementation of $And\_schema\ S\ S'$. The proof of this result in LEGO is almost 90 steps long, and again it must be regarded as incomplete since it makes use of the conjectures described above.

Proposition 2 (And_schema_intro)

$$\begin{array}{l}\forall S, S' : Schema . \forall str : Structure . \\ \quad (restricts\_to\_impl\ S\ str) \Rightarrow \\ \quad (restricts\_to\_impl\ S'\ str) \Rightarrow \\ \quad restricts\_to\_impl\ (And\_schema\ S\ S')\ str\end{array}$$

The next two results can be thought of as left and right elimination rules for schema conjunction. They have short proofs (less than 20 steps long) but these are again incomplete because they depend on outstanding conjectures.

Proposition 3 (And_schema_elim_left)

$$\begin{array}{l}\forall S, S' : Schema . \forall str : Structure . \\ \quad (Implements\ (And\_schema\ S\ S')\ str) \Rightarrow restricts\_to\_impl\ S\ str\end{array}$$

Proposition 4 (And_schema_elim_right)

$$\begin{array}{l}\forall S, S' : Schema . \forall str : Structure . \\ \quad (Implements\ (And\_schema\ S\ S')\ str) \Rightarrow restricts\_to\_impl\ S'\ str\end{array}$$

5 Conclusions and Future Work

The obvious next step in this work is to complete the proofs of the conjectures that remain outstanding. While these all appear to be obviously true, carrying out the formal proofs can take a long time. There is also a possibility that subtle problems may arise having to do with the way we use the powerful substitutive properties of inductive equality to get certain functions to typecheck. Finding out whether these problems arise and how they can be handled will be a test of the powers of UTT.

Next we would like to explore the usability of this encoding by carrying out a moderate-sized example. This will reveal whether we have made useful choices in defining, for instance, an implementation, and whether our theorems about schema operations are appropriate for doing structured program verification.

A major task would be to find out the relationship between our type-theoretical semantics for Z schemas and other proposed semantics. However even if we cannot do this in a formal way, we can still use examples to show how our encoding allows us to mimic the use of the Z notation within the LEGO proofchecker.

References

[BN92] Stephen Brien and John Nicholls. Z base standard, version 1.0. Electronically distributed, 1992.

[CM93] Claire Jones and Savi Maharaj. The LEGO Library. Electronically distributed, 1993.

[LP92] Zhaohui Luo and Robert Pollack. LEGO Proof Development System: user's manual. Technical Report ECS-LFCS-92-211, LFCS, Dept. of Computer Science, University of Edinburgh, 1992.

[Luo91b] Zhaohui Luo. A Unifying Theory of Dependent Types I. Technical Report ECS-LFCS-91-154, LFCS, Dept. of Computer Science, University of Edinburgh, 1991.

[Luo91a] Zhaohui Luo. Program Specification and Data Refinement in Type Theory. Technical Report ECS-LFCS-91-131, LFCS, Dept. of Computer Science, University of Edinburgh, 1991.

[Luo93] Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press (forthcoming), 1993.

[Mah90] Savitri Maharaj. Implementing Z in LEGO. Master's thesis, University of Edinburgh, 1990.

[Spi88] J.M. Spivey. *The Z notation: a reference manual.* Prentice-Hall International, 1988.

The Expressive Power of Structural Operational Semantics with explicit assumptions

Marino Miculan‡
Dipartimento di Matematica e Informatica
Università di Udine, Italy
miculan@di.unipi.it

Abstract

We explore the expressive power of the formalism introduced in [BH90] for defining the operational semantics of programming languages. This formalism derived from the Natural Semantics of Despeyroux and Kahn [Des86, Kah87] and arises if we take seriously the possibility of deriving assertions in Natural Semantics under assumptions, i.e. using hypothetic-general premises in the sense of Martin-Löf ([Mar84]). We investigate to what extent we can reduce to hypothetical premises the notions of store and environment of Plotkin's Structural Operational Semantics. We use this formalism to define the semantics of a functional language which features commands, blocks, procedures, complex declarations, structures and Abstract Data Types. We give the NOS style together with the denotational semantics and prove the adequacy of the former w.r.t. the latter. Moreover, we solve some other difficulties which arose in the previous treatment of variables in connection with procedures ([BH90]).

Natural Operational Semantics can be easily encoded in formal systems based on λ-calculus type-checking, such as the Edinburgh Logical Framework. We briefly investigate this and discuss some of the design choices.

1 Introduction

In order to establish formally properties about programs, we have to represent formally their operational semantics. A very successful style of presenting operational semantics is the one introduced by Gordon Plotkin and known as *Structural Operational Semantics* (SOS) ([Plo81]). The idea behind this approach is that all computational elaboration and evaluation processes can be constructed as logical processes and hence can be reduced to the sole process of formal logical derivation within a formal system.

For example, the SOS of a functional language is a formal system for inferring assertions such as $\rho \vdash M \to m$, where m is the *value* of the *expression* M, and ρ is the *environment* in which the evaluation is performed – usually a function mapping identifiers to values. The intended meaning of this proposition is "in the environment ρ, the evaluation of M gives m".

This style of specification does not have many of the defects of other formalisms (such as automata and definitional interpreters), since it is syntax-directed, abstract and easy to understand. This style of specification was proved to be very successful in various areas of

‡Work partially supported by EEC grant, BRA No. 6453 "Types for Proofs and Programs"

theoretical computer science. It was studied in depth by Kahn and many of his coworkers, and it has been used by Milner with the name of *Relational Semantics*. Nevertheless, the explicit presence of environments in propositions can exhibit several inconveniences:

- the abstraction power is limited: a function which maps identifiers to values amounts to von Neumann's computer's memory, and each assertion can predicate of only one memory at a time.

- Modularity is limited: if we extend the language by adding another kind of identifiers and denotable object (e.g. procedure identifiers and procedures), we have to introduce another environment function. Therefore, we have to change the assertion into the form $\rho, \tau \vdash M \to m$ where τ denotes the procedure declaration environment. So, all previous rules and derivations are not compatible any more with the new assertion.

- The system lacks conciseness. Environments appear in all rules but are seldom explicitly used, e.g. in the following rule:

$$\frac{\rho \vdash N_1 \to n_1 \quad \rho \vdash N_2 \to n_2}{\rho \vdash N_1 + N_2 \to plus(n_1, n_2)}$$

  Here ρ plays no rôle: it is merely transferred from conclusion to premises (in a top-down proof development). The environment is effectively used only when we are dealing with identifiers, i.e. when we either declare an identifier or evaluate it. These rules are e.g.

$$\frac{\rho \vdash M \to m \quad [x \mapsto n]\rho \vdash N \to n}{\rho \vdash \mathbf{let}\ x = M\ \mathbf{in}\ N \to n} \qquad \rho \vdash x \to \rho(x).$$

- It is well-known that in order to reason formally about properties of the operational semantics, it is necessary to encode the formal system into some proof-editor/checker. However, in most of the proof editors and checkers, encoding functions (such as the environments) can be rather cumbersome.

A possible solution to these drawbacks is the *Natural Operational Semantics* formalism (NOS) introduced in [BH90] as a refinement of the Natural Semantics originally proposed by Kahn and his coworkers ([Des86, Kah87]). This formalism arises if we take seriously the possibility of deriving under assumptions assertions in Natural Semantics, i.e. using hypothetic-general premises in the sense of Martin-Löf ([Mar84]). It is based in fact on Gentzen's Natural Deduction style of proof ([Gen69]): hypothetical premises are used to make assumptions about the values of variables. We investigate to what extent we can reduce to hypothetical premises the notions of store and environment of Plotkin's Structural Operational Semantics. Thus, instead of evaluating an expression within an environment, we compute its value under some assumptions about the values of its free variables. I.e. we replace explicit environments with implicit contextual structures, that is the hypothetical premises in Natural Deduction.

Going back to the previous functional language, it has two syntactic classes, *Expr*, the class of expressions (ranged over by M, N), and *Id*, the class of identifiers (ranged over by x, y), the former including the latter. The assertions of the formal system can be simplified to those of the form $M \Rightarrow m$, whose reading is "the value of expression M is m". There are no more contextual structures: the predicate is $\Rightarrow \subset Expr \times Expr$.

These assertions can be inferred by using a Natural Deduction style proof system, that is a set of rules of the form

$$\frac{\begin{array}{c}(\Delta_1)\\ \vdots\\ M_1 \Rightarrow m_1\end{array} \quad \dots \quad \begin{array}{c}(\Delta_k)\\ \vdots\\ M_k \Rightarrow m_k\end{array}}{M \Rightarrow m}\,\text{(possible side-condition)}$$

where the sets of assertions $\Delta_1, \dots \Delta_k$ are the *discharged assumptions*. Therefore, the evaluation of the expression M to the value m can be represented by the following derivation in N.D. style:

$$\begin{array}{c}\Gamma = \{x_1 \Rightarrow n_1, \dots, x_k \Rightarrow n_k\}\\ \mathcal{D}\\ M \Rightarrow m\end{array} \qquad \text{written} \quad \mathcal{D} : \Gamma \vdash M \Rightarrow m$$

where the hypotheses $\Gamma = \{x_1 \Rightarrow n_1, \dots, x_k \Rightarrow n_k\} (k \geq 0)$ can be interpreted as a set of variable bindings: the value of the variables involved in the evaluation of M. The meaning of this derivation is "in every environment which satisfies the assumptions in Γ, M is evaluated to m." This means that, given an environment ρ s.t. $\forall (x \Rightarrow m) \in \Gamma : \rho(x) = m$, there is a derivation of $\rho \vdash M \rightarrow m$ in the corresponding SOS proof system. Moreover, an assumption about the value of a variable can be discharged when it is valid only locally to a subcomputation. E.g. in the case of local declarations, in order to evaluate $\mathbf{let}\ x = N\ \mathbf{in}\ M$, we can evaluate M assuming that the value of x is the same as that of N. However, this extra assumption is inconsistent when we evaluate $\mathbf{let}\ x = N\ \mathbf{in}\ M$. Therefore, the **let** rule is in which the whole N.D. style appears

$$\frac{N \Rightarrow n \quad \begin{array}{c}(x \Rightarrow n)\\ \vdots\\ M \Rightarrow m\end{array}}{\mathbf{let}\ x = N\ \mathbf{in}\ M \Rightarrow m}$$

whose reading is "if n is the value of N and, assuming the value of x is n then m is the value of M, then the value of $\mathbf{let}\ x = N\ \mathbf{in}\ M$ is m."

Unfortunately the situation is not so simple, since this extra assumption can clash with a previous assumption on x which is valid globally. In order to deal with the issue of locality of variables we need to make a slight technical extension of original Gentzen's Natural Deduction style which allows us to reason directly on α-equivalence. This will be discussed in detail in sec.2. The above **let** rule is not correct (it allows us to evaluate $\mathbf{let}\ x = 0\ \mathbf{in}\ \mathbf{let}\ x = 1\ \mathbf{in}\ x$ to 0).

This truly N.D. approach has the benefit that all the rules which do not refer directly to identifiers appear in a simpler form than those in SOS style. There are no environment. For instance, the rule for the "+" function above mentioned becomes the following:

$$\frac{N_1 \Rightarrow n_1 \quad N_2 \Rightarrow n_2}{N_1 + N_2 \Rightarrow plus(n_1, n_2)}$$

In this paper, we address the following question: what kind of programming languages can be treated conveniently using this formalism. We are interested to understand to what extent we can reduce to assumptions the concepts of store, environment, binding and similar linear datatypes.

2 Analysis of the NOS style

In this section we try to convey briefly to the reader the main features of operational semantics in N.D. style. Recall that a N.D. style rule is as follows:

$$\frac{\begin{array}{ccc}(\Delta_1) & \dots & (\Delta_k)\\ \vdots & & \vdots \\ A_1 & \dots & A_k\end{array}}{A}$$

where $A, A_1, \dots, A_k$ are propositions and $\Delta_1, \dots, \Delta_k$ sets of propositions. More formally, a N.D. rule can be viewed as a concise description of a special kind of rule for deriving metapropositions of the form $\Gamma \vdash A$, the *sequents* ([Gen69, Avr91]). The above rule can be written in fact as follows:

$$\frac{\Gamma, \Delta_1 \vdash A_1 \quad \dots \quad \Gamma, \Delta_k \vdash A_k}{\Gamma \vdash A}$$

where Γ is any set of proposition. This rule means that, in order to prove that A is a consequence of a given Γ, we have to prove for $i = 1 \dots k$ that A_i is a consequence of Γ, Δ_i. In other words, for proving each A_i we can use some local assumptions Δ_i, the global hypotheses Γ always remaining valid. Therefore, the hypotheses of the sub-derivations of a derivation $\mathcal{D} : \Gamma \vdash A$ always contain Γ. This fact is at the core of the issues discussed in the following subsections.

2.1 The issue of local variables

Since the NOS rules are N.D. rules, if we have the following deduction $\mathcal{D} : \Gamma \vdash M \Rightarrow m$

$$\frac{\begin{array}{ccc}\Gamma, (\Delta_1) & & \Gamma, (\Delta_k)\\ \mathcal{D}_1 & \dots & \mathcal{D}_k \\ M_1 \Rightarrow m_1 & & M_k \Rightarrow m_k\end{array}}{M \Rightarrow m}$$

all the bindings in Γ are available in evaluating M_i, for $i = 1 \dots k$. As a consequence of this, the **let** rule showed in sec.1 above, is incorrect because previous (global) assumptions on locally defined variables can be used during the subevaluation, e.g. as follows:

$$\frac{0 \Rightarrow 0 \qquad \dfrac{1 \Rightarrow 1 \quad (x \Rightarrow 0)_{(1)}}{\mathbf{let}\ x = 1\ \mathbf{in}\ x \Rightarrow 0}}{\mathbf{let}\ x = 0\ \mathbf{in}\ \mathbf{let}\ x = 1\ \mathbf{in}\ x \Rightarrow 0}(1)$$

In order to overcome this problem we could use *higher-order syntax* à la Church. This technique originated with Church's idea to analyze $\forall x.P$ as $\forall(\lambda x.P)$ where $\forall$ has a higher order functionality: $\forall : (\mathit{Individuals} \to \mathit{Propositions}) \to \mathit{Propositions}$. It was further used by Martin-Löf and thoroughly expanded in the Edinburgh Logical Framework. See [AHM87] for a treatment of this in the context of λ-calculus and [Han88, MP91] of functional languages. For example, the construct **let** $x = M$ **in** N could be compiled to **let** $(\lambda x.N)M$, where **let** : $(\mathit{Expr} \to \mathit{Expr}) \to \mathit{Expr} \to \mathit{Expr}$. But this approach needs a form of textual substitution of an expression within another expression which cannot be expressed purely in Natural Deduction. Furthermore, the higher-order syntax cannot be used directly in the use of languages with imperative features. See [AHM87] for difficulties in handling Hoare's logic. In fact, it easily yields semantic inconsistencies, since it treats identifiers as the same of expressions.[1]

The difficulty of avoiding the capturing of local variables can be overcome in another way, by introducing an extension of original Gentzen's style of Natural Deduction called *α-notation* which explicitly manipulates textual substitution ([BH90]). In evaluating **let** $x = N$ **in** M, we have to replace all the occurrences of x in M with a new identifier never used before, say x', which will be bound to the value of n.

Textual substitution of identifiers can be represented by using two syntactic constructors, $\alpha, \underline{\alpha} : Id \times Id \times Expr \to Expr$. They are dual to one another and capture the notion of α-equivalence: $\alpha_{x/y}$ is an explicit denotation of the textual substitution of occurrences of y by x. Their meaning is defined by two sets of rules (rule schemas),

$$\Re : \frac{C[N]}{C[\alpha_{y/x}M]} \qquad \underline{\Re} : \frac{C[M]}{C[\underline{\alpha}_{x/y}N]}$$

where $C[\,]$ is any context (formula with hole), $x, y \in Id$ are identifiers, and M, N are expressions such that N is obtained from M by replacing *all* the occurrences of x by y.

Using this extension, the **let** rule is definable as follows:

$$\frac{N \Rightarrow n \quad \begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ \alpha_{x'/x}M \Rightarrow \underline{\alpha}_{x'/x}m\end{array}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}\; x' \text{ is a new variable}$$

where "x' is a new variable" means that x' appears neither in n, x, M e m nor in any assumption different from $(x' \Rightarrow n)$, which are discharged by the rule application. This side condition will be shown to be natural and easily implementable by a general hypothetical judgment in LF.

Actually, the evaluation of $\alpha_{x'/x}M$ can be performed by using only the α-rules, as follows:

$$\frac{N \Rightarrow n \qquad \dfrac{\dfrac{\begin{array}{c}(x' \Rightarrow n)_{(1)}\\ \vdots \\ M' \Rightarrow m'\end{array}}{M' \Rightarrow \underline{\alpha}_{x'/x}m}}{\alpha_{x'/x}M \Rightarrow \underline{\alpha}_{x'/x}m}}{\textbf{let } x = N \textbf{ in } M \Rightarrow m}\,(1)$$

where in M', m' all the occurrences of x have been replaced by x'. Therefore, any previous assumption about x cannot be used in evaluating M'.

This treatment of local variables correctly obeys the standard *stack discipline*: when we have to define a local variable, we allocate a new cell (represented by x', a new variable) where we store the local value (this is achieved by assuming $x' \Rightarrow n$). This allocation is active only during the evaluation of M (the derivation tree of $\alpha_{x'/x}M \Rightarrow \underline{\alpha}_{x'/x}m$); then, the cell is disposed (x' does not occur in any other place). The effect of $\underline{\alpha}$ is akin to that of a garbage collector and it is necessary to maintain the locality of x'.

[1]E.g, the application of the λ-abstraction containing a command **lambda** $\lambda x.[x := x + 1]x$ to 0 would be reduced to the evaluation of $[0 := 0 + 1]0$, which is meaningless.

2.2 What informations can assumptions represent?

The monotonicity of hypothesis structural rules implicit in N.D. rules has another immediate consequence: we can reduce to assumptions only informations which can be dealt with using a stack discipline. In particular, an side-effect assignment of pointers which induces variables aliasing (or sharing) is difficult to encode, since we would then necessitate of a vector. In fact, we cannot retrace given a hypothesis all the bindings which are involved on the shared variables whenever one of them changes its value.

However, in languages which do not allow sharing, assignments can be reduced to definitions of new variables. Therefore, we focus on this kind of languages, that is those whose semantics can be defined without using both environment and store. These comprise all purely functional languages, but also some interesting extensions of these which have genuinely imperative features. This is in fact our thesis: only languages whose denotational semantics is definable by using only the notion of environment can be conveniently handled by using NOS. In the following we describe some of these languages.

3 The language $\mathcal{L}_P$

In this section, we examine a functional language extended with imperative features as assignments which give it an imperative flavor. Its semantics can been successfully described by using NOS. We give its syntax, its NOS and denotational semantics and we prove that the former is adequate w.r.t. the latter. Finally, we will discuss the relation between the NOS and a SOS description of $\mathcal{L}_P$.

3.1 Syntax

$\mathcal{L}_P$ is an untyped λ-calculus extended by a set of structured commands. These commands are embedded into expressions using the "modal" operator $[\mathbf{on}\ \cdot\ \mathbf{do}\ \cdot]\cdot$. The expression $[\mathbf{on}\ x_1 = M_1; \ldots x_k = M_k\ \mathbf{do}\ C]M$ can be read as

> execute C in the environment formed only by the bindings $x_1 = M_1; \ldots; x_k = M_k$; use resulting values of these identifiers $x_1 \ldots x_k$ to extend the global environment in which M has to be evaluated, obtaining the value of the entire expression.

C cannot have access to "external" variables other than $x_1 \ldots x_k$, so all possible side effects are concerned with only these variables. Moreover, the entire **on-do** expression above does not have any side effect: all environment changes due to C's execution are local to M.

$\mathcal{L}_P$ allows us to declare and use procedures. For the sake of simplicity, but w.l.o.g., these procedures will take exactly two arguments. The first argument is passed by value/result, the second by value ([Plo81]). Furthermore, the body of a procedure cannot access global variables, but only its formal parameters (and locally defined identifiers, of course). This means that when $P(x, M)$ is executed within the scope of the declaration $\mathbf{proc}\ P(y, z) = C\ \mathbf{in}\ D$, C is executed in the environment formed by only two bindings: $\{y \Rightarrow n, z \Rightarrow m\}$, where n, m are the values of x, M respectively. After C's execution, the new value of y is copied back into x. So, $P(x, M)$ can effect only x.

The restriction on global access forbids sharing of identifiers, so there is no need of a store. This does not reduce drastically the expressiveness of the imperative language. Donahue has

Syntactic class *Id*

$x ::= i_0 \mid i_1 \mid i_2 \mid i_3 \mid \ldots$

Syntactic class *Expr*

$$\begin{aligned} M ::=\ & 0 \mid \mathit{succ} \mid \mathit{plus} \mid \mathit{true} \mid \mathit{false} \mid \leq \\ & \mid \mathit{nil} \mid M :: N \mid \mathit{hd} \mid \mathit{tl} \\ & \mid \mathbf{lambda}\, x.M \mid MN \\ & \mid \mathbf{let}\ x = M\ \mathbf{in}\ N \\ & \mid \mathbf{letrec}\, f(x) = M\ \mathbf{in}\ N \\ & \mid [\mathbf{on}\ R\ \mathbf{do}\ C]M \end{aligned}$$

Syntactic class *ProcId*

$P ::= p_1 \mid p_2 \mid p_3 \mid \ldots$

Syntactic class *Declarations*

$R ::= \langle\rangle \mid x = M; R$

Syntactic class *Commands*

$$\begin{aligned} C ::=\ & x := M \mid C; D \mid \mathbf{while}\ M\ \mathbf{do}\ C \\ & \mid \mathbf{nop} \mid \mathbf{if}\ M\ \mathbf{then}\ C\ \mathbf{else}\ D \\ & \mid \mathbf{begin\ new}\ x = M;\ C\ \mathbf{end} \\ & \mid \mathbf{proc}\ P(x, y) = C\ \mathbf{in}\ D \mid P(x, M) \end{aligned}$$

Figure 5: The syntax of $\mathcal{L}_P$

shown that in this case, the call-by-value/result is a "good" simulation of the usual call-by-reference ([Don77]).

In [BH90] a different definition of procedure is given. There, procedures parameters are passed only by value, but procedures can have access to global variables. However, there is a problem with this approach, since the N.D. treatment of procedures does not immediately lend itself to support side effects on global variables. That approach does not work; for instance, the expression $[\mathbf{on}\ x = 0\ \mathbf{do}\ \mathbf{proc}\ P(z) = (x := z)\ \mathbf{in}\ x := 1; P(\mathit{nil})]x$ would be evaluated to 1 instead of *nil*. This is due to the fact that the assignment made by $P(\mathit{nil})$ on the global variable x is local to the environment of the procedure itself. In fact, executions of such procedures leave the global environment unchanged.

3.2 Natural Operational Semantics

The complete NOS formal system for $\mathcal{L}_P$ consists of 70 rules; it appears in appendix A.1. Here, we can describe only some of its features; for a more extended discussion see [Mic92].

In order to perform evaluations and applications of λ-closures, command checking and execution, procedure checking and bookkeeping, we need to introduce some new constructors besides those of sec.3.1 and new predicates besides $\Rightarrow$. As in [BH90] the use of these new constructors is reserved: a programmer cannot directly utilize these constructors to write down a program. Below we list the constructors and predicates, and we briefly describe the most important ones.

Constructor	Functionality	
$[\_/\_]\_$	: $\mathit{Expr} \times \mathit{Id} \times \mathit{Expr}$	$\rightarrow \mathit{Expr}$
$\alpha, \underline{\alpha}$	: $\mathit{Id} \times \mathit{Id} \times \mathit{Expr}$	$\rightarrow \mathit{Expr}$
$\_ \cdot \_$	: $\mathit{Expr} \times \mathit{Expr}$	$\rightarrow \mathit{Expr}$
$[\_\vert\_]\_$	: $\mathit{Declarations} \times \mathit{Commands} \times \mathit{Expr}$	$\rightarrow \mathit{Expr}$
$[\_/\_]_{c}\_$	: $\mathit{Commands} \times \mathit{Id} \times \mathit{Commands}$	$\rightarrow \mathit{Commands}$
lambda	: $\mathit{Id} \times \mathit{Id} \times \mathit{Commands}$	$\rightarrow \mathit{Procedures}$
$[\_/\_]_{pe}\_$	: $\mathit{Procedures} \times \mathit{ProcId} \times \mathit{Expr}$	$\rightarrow \mathit{Expr}$
$[\_/\_]_{pc}\_$	: $\mathit{Procedures} \times \mathit{ProcId} \times \mathit{Commands}$	$\rightarrow \mathit{Commands}$

where *Procedures* is a new syntactic class defined as follows: $q ::= \textbf{lambda}\, x, y.C$

Judgment	Type	Judgment	Type
$\Rightarrow$	$\subset Expr \times Expr$	$\Rightarrow_p$	$\subset ProcId \times Procedures$
$value$	$\subset Expr$	$free_e$	$\subset Expr \times IdSet$
$closed$	$\subset Expr$	$free_c$	$\subset Commands \times IdSet$
$closed_p$	$\subset ProcId$	$\rhd$	$\subset Declarations \times IdSet$

where *IdSet* is the subset of *Expr* defined as follows: $I ::= nil \mid x \mid I_1 :: I_2$

substitution: the intuitive meaning of $[n/x]M$ is "the expression obtained from M by replacing all free occurrences of x with n." Just as for the **let** discussed in sec.2, in order to evaluate $[n/x]M$ we have to evaluate M under the assumption that the value of x is n, and hence any previous assumption on x must be ignored. This is implemented by the substitution rule, no.4, which, of course, is very similar to the **let** rule of sec.2:

$$\frac{value\ n \quad \begin{array}{c}(x' \Rightarrow n)\\ \vdots \\ \alpha_{x'/x} M \Rightarrow \underline{\alpha}_{x'/x} m\end{array}}{[n/x]M \Rightarrow m}\ x' \text{ is a new } Id \tag{4}$$

This rule is the core of the evaluation system. Many other evaluation rules, e.g. the one for **let**, are reduced to the substitution evaluation (rule no.6).

In NOS, to each sort of identifiers and substitution operators (e.g. *Id* and $[\_/\_]\_$, *ProcId* and $[\_/\_]_{pe}\_$, *Id* and $[\_/\_]_{c}\_$, etc.) there corresponds a specific substitution rule, similar in shape to rule no.4, which defines how to evaluate substitutions. In fact, this mechanism is used whenever one has to deal with standard static scoping. One can even think of these rules as a polymorphic variant of the same set of rules. Of course, minor adjustments have to be accounted for (rules no.29, no.39, no.35). More details can be found in [BH90, Mic92].

The operator $[\_/\_]\_$ is also used to record local environments in those values that are **lambda**-abstractions, i.e. the *closures*. An expression like $\textbf{lambda}\, x.M$ is evaluated into $[n_1/x_1]\ldots[n_k/x_k]\textbf{lambda}\, x.M$, where $x_1, \ldots, x_k$ are all the free identifiers of M but x, and $n_1, \ldots, n_k$ are their respective values. The construction of this closure is performed by rules no.7 and no.8; its application by rules no.10 and no.11

command execution: the intuitive meaning of $[R|C]M$ is the same as of $[\textbf{on}\ R\ \textbf{do}\ C]M$. This expression is introduced in order to apply the declaration R until it is empty (rule no.21); then, the command C is executed. We will write $[C]M$ instead of $[\langle\rangle|C]M$. It is interesting to notice that the assignment rule (no.22) and the **let** rule (no.6) are almost the same.

The judgment *value* encodes the assertion that an expression is a value, and so it cannot be further reduced nor its meaning is affected by an α-substitution.

The judgments *closed*, $closed_p$ are used during closure construction, in order to determine the bindings that we have to record (rules no.7, no.8, no.37). Informally, we can derive *closed* M if and only if M has no free variables. For its formal meaning, see th.1. The judgment *closed* belongs to static semantics: it can be inferred without using evaluation rules.

The judgments *free*, $free_c$, $\triangleright$ are used to check that command expressions [**on** D **do** C]M and procedures do not access global variable. Informally, we can infer $\Gamma \vdash free\ M\ I$ if and only if all the free variables of M appear in the list I (see th.2). On the other hand, $\triangleright$ collects the variables defined by a declaration D into a set (represented by a list of identifiers).

The judgment $\Rightarrow_p$ is used for bookkeeping the bindings between procedure identifiers and procedural abstraction (see rules no.34, no.36).

3.3 Denotational Semantics

In appendix B.1 we give the denotational semantics for the language $\mathcal{L}_P$. Domains are introduced to represent all the entities we have defined. This semantics is self-explanatory. We follow the usual syntax ([Sch86]); $\underline{\lambda}$ denotes the *strict abstraction*: for each meta-expression M with free variable x on pointed domain D, $(\underline{\lambda} x.M)\bot = \bot$. Furthermore, $\underline{\underline{\lambda}}$ is the *double-strict abstraction*: for each meta-expression $M \neq \bot$ with free variable x on pointed domain D with both $\bot$ and $\top$, $(\underline{\underline{\lambda}} x.M)\bot = \bot, (\underline{\underline{\lambda}} x.M)\top = \top$.

Moreover we use the standard domains without give their definition. The domains used are *Unit* (the set composed by only one point), $\mathbb{T}$ (the boolean set composed by two points, *true* and *false*), $\mathbb{N}$ (the set of natural numbers).

3.4 Adequacy

In this section we will show that the NOS description of $\mathcal{L}_P$ appearing in appendix A.1 is adequate w.r.t. the denotational semantics; that is, we will give soundness and completeness results of one semantics w.r.t. the other. We will only sketch the proofs; for further details see [Mic92].

3.4.1 Soundness

Definition 1 *A set of formulae* Γ *is a* canonical hypothesis *if*

- *it contains only formulae like "*$x \Rightarrow n, P \Rightarrow_p q, closed(x), closed_p(P)$*";*
- *if* $x \Rightarrow n, x \Rightarrow m \in \Gamma$ *then* m *and* n *are syntactically the same expression;*
- *if* $P \Rightarrow_p q, P \Rightarrow_p q' \in \Gamma$ *then* q *and* q' *are syntactically the same procedure abstraction;*

where $x \in Id, P \in ProcId$ *and* $m, n \in Expr$, $q, q' \in Procedures$.

In the rest of section, Γ will denote a generic canonic hypothesis.

Definition 2 *Let* G *be a formula. With* $\Gamma \vdash G$ *we denote the N.D. derivation of* G*, whose undischarged assumptions are in* Γ.

Definition 3 *For* $M \in Expr$, $\mathsf{FV}(M) \subset Id \cup ProcId$ *is the* set of free identifiers of M.

The definition of FV can be naturally extended to *Commands*, bearing in mind that $\mathsf{FV}(x := M) \stackrel{\text{def}}{=} \mathsf{FV}(M)$.

Definition 4 *For any declaration* $R \in Declarations$, $\mathsf{DV}(M) \subset Id$ *is the* set of variables defined by R *and it is defined as* $\mathsf{DV}(x_1 = M_1; \ldots; x_k = M_k) \stackrel{\text{def}}{=} \{x_1, \ldots, x_k\}$.

Definition 5 *The set of* Γ-closed identifiers $\mathsf{C}(\Gamma)$ *is* $\mathsf{C}(\Gamma) \stackrel{\text{def}}{=} \{x \in \mathit{Id} \mid \mathit{closed}(x) \in \Gamma\} \cup \{P \in \mathit{ProcId} \mid \mathit{closed}(P) \in \Gamma\}$.

Theorem 1 (Adequacy of *closed*) $\forall \Gamma, \forall M \in \mathit{Expr} : \Gamma \vdash \mathit{closed}\ M \iff \mathsf{FV}(M) \subseteq \mathsf{C}(\Gamma)$

Proof. $\Longrightarrow$ By induction on the structure of derivation $\Gamma \vdash \mathit{closed}\ M$.
$\Longleftarrow$ By induction on the syntactic structure of the expression M. ∎

Theorem 2 (Adequacy of *free*) $\forall \Gamma, \forall m \in \mathit{Expr}, \forall C \in \mathit{Commands}, \forall l \in \mathit{IdSet}$:

$$\begin{aligned} \Gamma \vdash \mathit{free}\ m\ l &\iff \mathsf{FV}(m) \cap \mathit{Id} \subseteq l \wedge \mathsf{FV}(m) \cap \mathit{ProcId} \subseteq \mathsf{C}(\Gamma) \\ \Gamma \vdash \mathit{free}\ C\ l &\iff \mathsf{FV}(C) \cap \mathit{Id} \subseteq l \wedge \mathsf{FV}(C) \cap \mathit{ProcId} \subseteq \mathsf{C}(\Gamma) \end{aligned}$$

Proof. By suitable inductions. ∎

Definition 6 *Let* $I \subseteq \mathit{Id} \cup \mathit{ProcId}$ *and let* $\rho, \rho' : \mathbb{E}$ *be two environments. We say that* ρ *and* ρ' agree on I *(*$\rho \equiv_I \rho'$*) if the following properties hold:* $\forall x \in I \cap \mathit{Id} : (\mathit{access}\ [\![x]\!]\ \rho = \mathit{access}\ [\![x]\!]\ \rho')$ *and* $\forall P \in I \cap \mathit{ProcId} : (\mathit{procaccess}\ [\![x]\!]\ \rho = \mathit{procaccess}\ [\![x]\!]\ \rho')$.

Note that all environments ρ, ρ' agree on the empty set, that is $\forall \rho, \rho' \in \mathbb{E} : \rho \equiv_{\emptyset} \rho'$.

Lemma 1 *Let be* $m \in \mathit{Expr}$, $R \in \mathit{Declarations}$, $\rho, \rho' \in \mathbb{E}$. *Then* $\rho \equiv_{\mathsf{FV}(m)} \rho' \Rightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'; \rho \equiv_{\mathsf{FV}(C)} \rho' \Rightarrow \mathcal{C}[\![C]\!]\rho = \mathcal{C}[\![C]\!]\rho'; \rho \equiv_{\mathsf{FV}(R)} \rho' \Rightarrow \mathcal{D}[\![R]\!]\rho = \mathcal{D}[\![R]\!]\rho'$

Proof. By simultaneous induction on the syntactic structure of expressions, commands and declarations. ∎

Theorem 3 $\forall \Gamma, \forall \rho, \rho' \in \mathbb{E}, \forall m \in \mathit{Expr}, \forall l \in \mathit{IdSet}$, *if* $\rho \equiv_{\mathsf{C}(\Gamma)} \rho'$ *then*

$$\Gamma \vdash \mathit{closed}\ m \Longrightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'; \qquad (\Gamma \vdash \mathit{free}\ C\ l) \wedge \rho \equiv_{\mathit{leaves}(l)} \rho' \Longrightarrow \mathcal{C}[\![C]\!]\rho = \mathcal{C}[\![C]\!]\rho'$$

Proof. Apply th.1 to lemma 1. ∎

Definition 7 *We define* $\overline{\Gamma}$, *the* Γ closure, *as follows:* $\overline{\Gamma} \stackrel{\text{def}}{=} \Gamma \cup \{\mathit{closed}\ x \mid y \Rightarrow n \in \Gamma, x \in \mathsf{FV}(n)\} \cup \{\mathit{closed}_p\ P \mid y \Rightarrow n \in \Gamma, P \in \mathsf{FV}(n)\}$.

Lemma 2 $\forall x \Rightarrow n \in \Gamma : \overline{\Gamma} \vdash \mathit{closed}\ n$.

Proof. Trivial, by definition of $\overline{\Gamma}$ and th.1. ∎

Theorem 4 *Let* $m \in \mathit{Expr}$*; if* $\Gamma \vdash \mathit{value}\ m$ *then* $\overline{\Gamma} \vdash \mathit{closed}\ m$.

Proof. A tedious induction on the structure of derivations $\Gamma \vdash \mathit{value}\ m$. ∎
Note that the statement does not hold if we use Γ in place of $\overline{\Gamma}$. E.g., $y \Rightarrow x \vdash \mathit{value}\ x$ by rule no.1, but $y \Rightarrow x \not\vdash \mathit{closedx}$. The information we are missing is that x should be closed, since it appears on the right-hand side of $\Rightarrow$.

Definition 8 *A canonic hypothesis* Γ *is a* well-formed hypothesis (wfh) *if* $\Gamma = \overline{\Gamma}$.

In particular, $\emptyset$ is a wfh.

Theorem 5 $\forall \Gamma\, \mathit{wfh}, m \in \mathit{Expr}, \rho, \rho' \in \mathbb{E} : \Gamma \vdash \mathit{value}(m) \wedge \rho \equiv_{\mathsf{C}(\Gamma)} \rho' \Longrightarrow \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$.

Proof. Apply lemma 2 to th.4 remembering that $\Gamma = \overline{\Gamma}$ (it is wfh). ∎

Corollary 1 (Soundness of *value***)** $\forall m \in \mathit{Expr} : \vdash \mathit{value}\ m \Longrightarrow \forall \rho, \rho' \in \mathbb{E} : \mathcal{E}[\![m]\!]\rho = \mathcal{E}[\![m]\!]\rho'$

Proof. Just put $\Gamma = \emptyset$ in th.5. ∎

Definition 9 *Let Γ be a canonical hypothesis. We say that a $\rho \in \mathbb{E}$ is* Γ-compatible *(written* Γ-comp(ρ)*) if* $\forall (x \Rightarrow n) \in \Gamma : \mathit{access}\ [\![x]\!]\ \rho = \mathcal{E}[\![n]\!]\rho$*, and* $\forall P \Rightarrow q \in \Gamma : \mathit{procaccess}\ [\![x]\!]\ \rho = \mathcal{Q}[\![q]\!]\rho$

This is another place where the conciseness of the N.D. formalism comes into play. The domain of Γ-compatible environments can be much larger than the set of variables which occur on the left of assumptions in Γ.

Theorem 6 $\forall M, m \in \mathit{Expr}, \Gamma\ \mathit{wfh}, \rho \in \mathbb{E} : \Gamma\text{-comp}(\rho) \wedge \Gamma \vdash M \Rightarrow m \ \Rightarrow\ \mathcal{E}[\![M]\!]\rho = \mathcal{E}[\![m]\!]\rho$

Proof. By a long induction on the structure of derivations, using the previous results. ∎

Corollary 2 (Soundness of NOS wrt DS) $\forall M, m \in \mathit{Expr} : \vdash M \Rightarrow m \Longrightarrow \mathcal{E}[\![M]\!] = \mathcal{E}[\![m]\!]$

Proof. Just put $\Gamma = \emptyset$ in th.6, and notice that every environment is $\emptyset$-compatible. ∎

3.4.2 Completeness

A completeness result is something like an "inverse" of corollary 2. However, a statement inverse of corollary 2 cannot hold. E.g. for $M = m = (\mathbf{lambda}\, x.x)0)$ it is $\mathcal{E}[\![M]\!] = \mathcal{E}[\![m]\!]$ but of course $\not\vdash M \Rightarrow m$. In fact, only some expressions can appear as values. We need a new definition:

Definition 10 *Let $M \in \mathit{Expr}$. An hypothesis Γ is* suitable for M*, (M-suit(Γ)), if* $\forall x \in \mathsf{FV}(M) \exists (x \Rightarrow n) \in \Gamma$ *and* $\forall P \in \mathsf{FV}(M) \exists (P \Rightarrow_p q) \in \Gamma$*.*

In other words, an hypothesis is suitable for M if it contains enough bindings to evaluate M.

Theorem 7 *Let $M \in \mathit{Expr}$, Γ wfh and $\rho \in \mathbb{E}$. If $\mathcal{E}[\![M]\!]\rho \neq \bot, \top$ and* M-suit(Γ) *and* Γ-comp(ρ)*, then $\exists m \in \mathit{Expr} : \Gamma \vdash M \Rightarrow m$.*

Proof. The difficulty in the proof is the problem: given an expression M whose meaning, in a given environment, is a proper point of $\mathbb{V}$, and a suitable hypothesis Γ, we have to build up a deduction $\Gamma \vdash M \Rightarrow m$, for some m.[2] This cannot be done by induction on the syntactic structure of M, since our language is higher order; in fact, the evaluation of M can use M itself, and not only its subterms (see e.g. rule no.26). Nevertheless, we can prove the theorem by using the technique of *inclusive predicates*, developed by Milne and Plotkin. ∎

[2]By th.6, this m has the same meaning of M

3.5 Adequacy w.r.t. the SOS

In the previous subsection we have proved the adequacy of the NOS specification of $\mathcal{L}_P$ w.r.t. the denotational semantics. Actually, the same adequacy can be proved w.r.t. a Structural Operational Semantics (*à la Plotkin*, [Plo81]). One can define a complete "input-output" SOS system for $\mathcal{L}_P$, that is a system for deriving two kinds of judgments:

evaluation of expressions: $\rho \vdash_{SOS} M \to m$
execution of commands: $\rho \vdash_{SOS} C \to \rho'$

where ρ, ρ' are finite environments, i.e. they are defined on a finite number of identifiers, and m is a value. In such a SOS system, there is no problem in handling substitutions, since we merely update the environment function in the subderivation:

$$\frac{\rho[x \mapsto n] \vdash_{SOS} M \to m}{\rho \vdash_{SOS} [n/x]M \to m}$$

Of course, this is not a linearized Natural Deduction style system since we may delete a previous binding on x from the environment. However, the following results can be proved:

Theorem 8 (Soundness of NOS wrt SOS) *Let $M, m \in Expr$, Γ wfh and ρ finite environment. If $\forall (x \Rightarrow n) \in \Gamma : \rho(x) = n$ and $\Gamma \vdash M \Rightarrow m$, then $\rho \vdash_{SOS} M \to m$*

Theorem 9 (Completeness of NOS wrt SOS) *Let $M \in Expr$, Γ wfh and ρ finite environment. If $\rho \vdash_{SOS} M \to m$ and $\forall x \in \mathsf{FV}(M) : (x \Rightarrow \rho(x)) \in \Gamma$, then $\Gamma \vdash M \Rightarrow m$.*

Both theorems can be proved by using techniques similar to those of previous subsection. Moreover, the completeness result does not need the technique of inclusive predicates, but only a simpler structural induction on the derivation $\rho \vdash_{SOS} M \to m$.

4 Some remarks about language design

$\mathcal{L}_P$ is quite different from the language considered in [BH90]. There are several reasons for these changes. In some cases these are motivated by the desire to have a natural soundness result (see section 3.1 for remarks concerning procedures).

In our language, commands are embedded into expressions by the **on-do** construct. A simpler formalism for applying directly commands to expressions is used in [BH90], i.e. the "modal" operator $[\ ] : Commands \times Expr \to Expr$, so that $[C]M$ is an expression if $M \in Expr, C \in Commands$. Informally, the value of $[C]M$ is the value of M after the execution of C; C can affect any variable which is defined before its execution. Furthermore, as all expressions, $[C]M$ has no side-effects, that is evaluating $[C]M$ does not change the global environment any more than evaluating 0 or *nil*. C affects only the local environment which is used to evaluate M, but its side effects are not "filtered" by a declaration of accessible variables. In order to appreciate the difference in notation between the two approaches compare the following semantically equivalent expressions:

in the system of [BH90]: **let** $x = 0$ **in** $[x := nil]x$; in $\mathcal{L}_P$: [**on** $x = 0$ **do** $x := nil]x$

At first it seems that the latter is more complex and nothing has been gained. But the former expression might lead us to think that we can define functions with local state variables and more interesting expressions objects, but this is not the case! For instance, if we try to model a bank account defining a function `withdraw` which takes the amount to be withdrawn from the balance (an example snarfed from [AS85]):

```
let bal = 100; withdraw = lambda a.[bal:=bal-a]bal
in let remaining = (withdraw 50)
   in  (withdraw 30)
```

The system in [BH90] will evaluate it to 70 instead 20: the first withdraw has no effect. The reason is that in the closure of `withdraw`, `bal` is bound to 100, and this binding is reapplied to local environment whenever `withdraw` is applied; this "reinitializes" `bal` to 100 each time (see rules no.8, no.9, no.11). Therefore, an application of `withdraw` cannot affect any following application.

Thus, [BH90]'s system may lead to misunderstanding the meaning of some expressions. We decide to avoid this by writing explicitly the variables which a command can affect, and making explicit that such variables are always reinitialized whenever the command is executed. By writing $[\mathbf{on}\, x_1 = M_1; \ldots ; x_k = M_k\ \mathbf{do}\ C]M$ we immediately know that, *before* C is executed, the "interface variables" $x_1 \ldots x_k$ are initializated. Therefore, an obscure program, like the `withdraw` one, cannot be written in $\mathcal{L}_B$. In $\mathcal{L}_B$, the above `withdraw` function should be declared as follows:

```
let withdraw = lambda a.[on bal = 100 do bal:=bal-a]bal
in ...
```

and hence it is clear(er) what is the meaning of `withdraw`.

This aspect is however a major problem: neither in [BH90] system nor in $\mathcal{L}_P$ the `withdraw` function with the intended meaning of [AS85] can be written. We'll elaborate on this in sec.7.

5 Some extensions of $\mathcal{L}_P$

In this section we discuss some further extensions of $\mathcal{L}_P$ whose semantics can be expressed without stores because there is no variable sharing. These extensions concern complex declarations, structures and imperative modules. Due to lack of space, we can give only a brief description of these extensions. Their NOS and denotational semantics are in appendix A and B respectively. We deal with each extension by itself, by simply adding new rules to the formal system without altering the previous ones. This illustrates modularity of NOS which allows us to add new rules for new constructs without changing the previous ones. For each extension, one can prove adequacy of NOS w.r.t. the denotational semantics ([Mic92]). The soundness and completeness theorems and proofs also can been gradually extended by discussing only the new cases due to the extra rules.

5.1 Complex Declarations

$\mathcal{L}_D$ is obtained from $\mathcal{L}_P$ by adding expressions of the form **let** R **in** M where R is a complex declaration like in Standard ML ([Har89]). In spite of the syntactic simplicity of this extensions (fig.6), it appears to be unavoidable to define an entire evaluation system for declarations (rules no.77—93). The value of complex declarations are finite sets of bindings. These sets are represented by expressions called *syntactic environments*; they are trees whose leaves are of the form $x \mapsto n$ where $\mapsto$: $Id \times Expr \to Expr$ is a new local constructor. We need to introduce furthermore several constructors and a judgment for applying such syntactic environments to expressions and declarations ($\{\_\}\_$, $\{\_\}_d\_$) and for inferring expression closures ($\langle\_\rangle\_$, $\gg$). Informally, one can derive $\Gamma \vdash R \gg I$ iff all expressions contained in R are closed in Γ and I is the set of identifiers defined by R. On the other hand, $\Gamma \vdash$ *closed* $\langle I\rangle M$ iff all free variables in M

Syntactic class *Expr*
$M ::= \ldots \mid \textbf{let}\ R\ \textbf{in}\ M$

Syntactic class *Declarations*
$R ::= \ldots \mid R; S \mid R\ \textbf{and}\ S$

Figure 6: The syntax of $\mathcal{L}_D$

Syntactic class *LongId*
$u ::= x \mid u.x$

Syntactic class *Commands*
$C ::= \ldots \mid u := M$

Syntactic class *Expr*
$M ::= \ldots \mid \textbf{sig}\ x_1 \ldots x_k\ \textbf{end}$
$\quad \mid \textbf{struct}\ x_1 = M_1; \ldots ; x_k = M_k\ \textbf{end}$
$\quad \mid M : N \mid \textbf{open}\ u\ \textbf{in}\ M$

Figure 7: The syntax of $\mathcal{L}_{M_F}$.

but the ones in I are *closed* in Γ. Once the rules will be laid down, these fact will be formally provable. Using this set of rules, we can define precisely when a complex **let** is closed without using any evaluation, since *closed* is a property belonging to static semantics. An adequacy theorem similar to th.1 can be proved for the system given in sec.A.2. In [BH90] there is a simpler approach; it uses the complex declaration evaluation in order to determine the set of defined identifiers. This approach is not complete: there are closed expression whose *closed* property cannot be inferred in [BH90]'s system. E.g., $\textbf{let}\ o = (\textbf{lambda}\ x.xx); z = (oo)\ \textbf{in}\ z$.

5.2 Structures and signatures

$\mathcal{L}_{M_F}$ extends $\mathcal{L}_P$ by adding a module system like that of Standard ML ([Har89]), where a module is "an environment turned into a manipulable object". Like SML, a module (here called *structure*) has a *signature*, and we can do *signature matching* in order to "cast" structures. However, there are some differences between SML and $\mathcal{L}_{M_F}$. First, in $\mathcal{L}_{M_F}$ structures and signatures are indeed expression. Therefore, they may be associated to identifiers with simple **let**s, without using special constructs. These **let**s can appear anywhere in expressions, not only at top level. Structures and signatures can be manipulated by common functions; however, there are not *functors* since the sharing specification is not implemented.

The NOS should be self-explanatory.

5.3 Imperative modules (Abstract Data Types)

The extension $\mathcal{L}_{M_I}$ introduces modules *à la Morris* ([Mor73]). In this formulation, a module is very close to an Abstract Data Type: it contains

1. a set of local variables, recording the *state* of the module; they are not accessible from outside the module;

2. some code for the initialization of the local variables above;

3. a set of procedures and functions which operate on these local variables and are the only part accessible from outside the module (the *interface*).

Syntactic class *ModId*
$T ::= t_1 \mid t_2 \mid t_3 \mid \ldots$

Syntactic class *Expr*
$M ::= \ldots \mid T.f$

Syntactic class *Commands*
$C ::= \ldots \mid \mathbf{module}\ T\ \mathbf{is}$
$\quad x = M; \mathbf{proc}\ P(y) = C; \mathbf{func}\ f = N$
$\quad \mathbf{in}\ D \mid T.P(M)$

Figure 8: The syntax of $\mathcal{L}_{M_I}$.

From outside a module we can only evaluate its functions, which do not produce side-effects, and execute its procedures, which can modify the state of the module (the value of local variables). In order to illustrate the idea, but w.l.o.g., we discuss only modules with exactly one local variable, one procedure with one argument (passed by value) and one function (fig.8).

As for the previous languages, we do not need a representation of the store in defining the semantics of this kind of module ([Don77]). The rules for the specification of the imperative modules are certainly the most complex of those discussed in this paper. They are based on the principle of distributing as much as possible under the form of hypothetical assumption in deductions. In a module there are three informations: the state, the procedure and the function. Actually, only the state is subject to changes upon execution of the module procedure. We split these three informations and record them using three different judgments (see rule no.113). The predicates of these assumptions are the following:

$$\Rightarrow_m \subset \mathit{ModId} \times (\mathit{Expr} \times \mathit{ModId}) \quad \Rightarrow_{mp} \subset (\mathit{ModId} \times \mathit{ProcId}) \times \mathbb{Q} \quad \Rightarrow_{mf} \subset (\mathit{ModId} \times \mathit{Id}) \times \mathit{Expr}$$

We use a lot of syntactic sugar; for instance, instead of $\Rightarrow_{mp} ((T, P), \mathbf{lambda}\, x, y.C)$, we write $T.P \Rightarrow_{mp} \mathbf{lambda}\, x, y.C$.

When the state of a module changes (by executing its procedure), we have to substitute only the assumption involving $\Rightarrow_m$; the other two remain the same. Thus, while the procedure and the function are left associated to original module identifier, the state becomes associated to a new *ModId*, and this substitution affects a part of the declaration to be evaluated (see rules no.115 no.114). The link between the new state and the procedures is maintained by the module identifier which appears on right of $\Rightarrow_m$ assumption: it is merely copied from the old assumption into the new one (rule no.115).

When a module procedure has to be executed ($T.P(M)$), first we look for the state of the module T, by requiring $T \Rightarrow_m (p, T')$. Here we find the original module identifier, T'. The invoked procedure is then associated to this identifier in the assumption $T'.P \Rightarrow_{mp} \mathbf{lambda}\, x, y.C$. After having bound x and y respectively to module variable value (p) and actual parameter (m), we execute C and get back the new value of the state variable. Finally, we substitute T with the new module state.

Module function evaluation is similar to procedure call, but simpler (rule no.116).

We can successfully implement the bank account examined in sec.4 by using this kind of modules, e.g. as follows:

```
module account is
  bal = 100;
  proc withdraw(amount) = bal := bal - amount;
  func balance = bal
in ...
```

After this declaration, we can withdraw an amount A from the balance by executing the command `account.withdraw(`A`)`, and know how much money we have left by evaluating `account.balance`.

However, even this notion of module is too weak to adequately model "functions with local state" as are necessary, for instance, in realizing memoized functions. In fact, as soon as an instance of a module is packaged within a λ abstraction, its connection with its parent (definition) is severed.

6 Encoding NOS in LF

From a logician's point of view, the Natural Operational Semantics of a language is just a formal logical system in Natural Deduction style. Therefore, it can be easily encoded in interactive proof-checkers based on type-checking of typed λ-calculus, such as the Edinburgh Logical Framework (LF, [HHP93]). This was actually one of the main motivations for introducing and investigating the systems of this paper. A first outline about this can be found in [BH90]. In [Mic92] a complete encoding of the semantics of the **while** subset of $\mathcal{L}_P$ appears.

The LF encoding of NOS has several significant consequences. When we encode the operational semantics in LF, we have to discuss details that are normally left out or too often taken for granted or even "swept under the rug". For instance, in rule no.4 we require that "x is a new identifier", but we do not give a formal definition of this. When we encode NOS in LF, this condition has to be expressed formally and unambiguously.

Furthermore, we can use this encoding with *proof editors* based on LF, such as LEGO ([LPT89]), and *theorem provers*, such as Elf ([Pfe89]). LEGO can be successfully used to develop derivations (= computation traces) and to verify properties about the semantics themselves, e.g. equivalence between constructs. During the phase of operational semantics developing, we can try our rules and look for inconsistencies. Thus, we have immediately a powerful tool for semantic development and consistency checking.

On the other hand, theorem provers such as Elf can be used to get an interpreter prototype for free: immediately after we have encoded in Elf the LF representation, we can ask queries like `?- True(eval` M `V).` where M is (the encoding of) an expression. In resolving this goal, Elf instantiates `V` to M's value, and develops a term which represent the deduction $\vdash M \Rightarrow V$, that is the computation trace of the evaluation of M.

In these systems we can prove several meta-results about semantics. In [Mic92] the equivalence between two different NOS of the same **while**-language is developed. One of these semantics is "natural", clear but inefficient. The rules for the **while** execution are the following:

$$\frac{M \Rightarrow \mathit{true} \quad [C]([\mathbf{while}\, M \ \mathbf{do}\ C]N) \Rightarrow n}{[\mathbf{while}\, M \ \mathbf{do}\ C]N \Rightarrow n} \qquad \frac{M \Rightarrow \mathit{false} \quad N \Rightarrow n}{[\mathbf{while}\, M \ \mathbf{do}\ C]N \Rightarrow n}$$

This semantics needs to backtrack and to re-evaluate the test expression if it does not match the required value in the assumption. The second semantics overcomes this drawback by introducing an auxiliary judgment, *dowhile*:

$$\frac{M \Rightarrow_a m \quad \mathit{dowhile}\, m\, M\ C\ N \Rightarrow_a n}{[\mathbf{while}\, M \ \mathbf{do}\ C]N \Rightarrow_a n} \quad \frac{[C]([\mathbf{while}\, M \ \mathbf{do}\ C]N) \Rightarrow_a n}{\mathit{dowhile}\,\mathit{true}\, M\ C\ N \Rightarrow_a n} \quad \frac{N \Rightarrow_a n}{\mathit{dowhile}\,\mathit{false}\, M\ C\ N \Rightarrow_a n}$$

Here, backtracking and double evaluation are not needed any more (actually, the $_a$ means "algorithmic").

By using a technique used by Michaylov and Pfenning for functional languages ([MP91]), we can prove the equivalence between these two semantics by encoding in Elf a judgment, $\texttt{naeq} : \prod_{M,m \in Expr} (M \Rightarrow m) \rightarrow (M \Rightarrow_a m) \rightarrow \mathsf{Type}$. This judgment represents the equivalence between "natural" and "algorithmic" computation traces. Asking Elf about queries of the form `?- naeq` D D' where one of D, D' is instantiated to a derivation in one semantics, the system automatically gives us the equivalent derivation in the other semantics. In this way we have defined a bijection between the computation traces of the two semantics.

We can think the former semantic as the "theoretical" semantics of the language, and the latter as the real implementation. Thus, the formal equivalence proved in Elf between them can be seen as the backbone of a proof of compiler correctness.

7 Concluding remarks

In this paper we have described the expressive power of the Natural Operational Semantics formalism. We have seen that this formalism handles successfully languages which do not allow variable aliasing, or sharing, i.e. variables whose semantics does not necessitate of stores. We have shown some of these languages: functional languages extended with a restricted form of commands and procedures, blocks, complex declarations, modules *à la ML* (structures and signatures) and modules *à la Morris.*

This formalism improves abstractness and modularizability of Plotkin's Structural Operational Semantics and Kahn's Natural Semantics. Furthermore, such a operational description can be easily encoded in LF. Such encodings can be used within implementations of LF (LEGO and Elf), giving us powerful tools for developing language semantics formally, for checking correctness of translators and for proving semantic properties.

Unfortunately, so far we have not been able to give the semantics of a truly imperative language using this distributed formalism. It seems that one cannot give simultaneously a representation of the store by means of assumptions. Without encoding a store we cannot describe usual imperative phenomena like side-effects with aliasing, argument passage of parameters by-reference and so on. Therefore, this formalism seems not general enough to deal with expressions with side-effects, functions with local state variables or memoization, Pascal procedures, i.e. procedures with global variables and call-by-reference.

7.1 The aim: the NOS of ML

We have seen that the NOS style can describe the operational behavior of a language very close to ML. We think that exception handling can be added to $\mathcal{L}_{M_F}$ quite easily ([BH90]). The real lack of our languages w.r.t. ML is the absence of the store: ML is a store-based language. Therefore, in order to capture fully the semantics of ML (and encode it in LF) we have to find some representation of the store. This is a task remaining to be done. Currently we are investigating further extension of the Natural Deduction style similar to the α-notation. Ideally, we would like to extend the formalism as much as is needed to describe the semantics of a language like the following one:

$$M ::= x \mid 0 \mid succ \mid \mathbf{lambda}\, x.M \mid M\ N \mid \mathbf{mk}\, M \mid \mathbf{get}\, x \mid \mathbf{set}\, M\ N$$

where, roughly speaking, **mk** corresponds to ML's `ref`, **get** to `!` and **set** to `:=` ([Har89]). The NOS of this language should be easily extended to that of ML.

A Rules for the natural operational semantics

A.1 NOS of $\mathcal{L}_P$

A.1.1 Rules for judgment *value*

$$\frac{M \Rightarrow m}{value\ m} \quad (1)$$

$$\frac{}{value\ m} \quad (2)$$
m is a constant

A.1.2 Schemas for $\alpha, \underline{\alpha}$

$$\frac{C[J]}{C[\alpha_{j/i}I]} \qquad \frac{C[I]}{C[\underline{\alpha}_{i/j}J]} \quad (3)$$

where $C[\ ]$ is a context and $i, j \in Id, I, J \in Expr$ such that J is obtained from I by replacing occurrences of i with j.

A.1.3 Rules for judgment $\Rightarrow$

$$\frac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\\ value\ n \quad \alpha_{x'/x}M \Rightarrow \underline{\alpha}_{x'/x}m\end{array}}{[n/x]M \Rightarrow m} \quad (4)$$
x' is a new variable

$$\frac{value\ m}{m \Rightarrow m} \quad (5)$$

$$\frac{N \Rightarrow n \quad [n/x]M \Rightarrow m}{\textbf{let}\ x = N\ \textbf{in}\ M \Rightarrow m} \quad (6)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots\\ closed\ M\end{array}}{\textbf{lambda}\ x.M \Rightarrow \textbf{lambda}\ x.M} \quad (7)$$

$$\frac{\begin{array}{c}(closed\ y)\\ \vdots\\ y \Rightarrow n \quad \textbf{lambda}\ x.M \Rightarrow m\end{array}}{\textbf{lambda}\ x.M \Rightarrow [n/y]m} \quad (8)$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n \quad m \cdot n \Rightarrow p}{MN \Rightarrow p} \quad (9)$$

$$\frac{[n/x]M \Rightarrow p}{(\textbf{lambda}\ x.M) \cdot n \Rightarrow p} \quad (10)$$

$$\frac{value\ n \quad [m'/x](m \cdot n) \Rightarrow p}{([m'/x]m) \cdot n \Rightarrow p} \quad (11)$$

$$\frac{value\ n}{succ \cdot n \Rightarrow succ \cdot n} \quad (12)$$

$$\frac{value\ m}{plus \cdot m \Rightarrow plus \cdot m} \quad (13)$$

$$\frac{value\ n}{(plus \cdot 0) \cdot n \Rightarrow n} \quad (14)$$

$$\frac{(plus \cdot m) \cdot n \Rightarrow p}{(plus \cdot (succ \cdot m)) \cdot n \Rightarrow succ \cdot p} \quad (15)$$

$$\frac{\begin{array}{l}\textbf{let}\ f = (\textbf{lambda}\ x.\textbf{letrec}\\ \qquad f(x) = N\ \textbf{in}\ N)\ \textbf{in}\ M \Rightarrow p\end{array}}{\textbf{letrec}\ f(x) = N\ \textbf{in}\ M \Rightarrow p} \quad (16)$$

$$\frac{M \Rightarrow m \quad N \Rightarrow n}{M :: N \Rightarrow m :: n} \quad (17)$$

$$\frac{value\ m}{hd \cdot (m :: n) \Rightarrow m} \quad (18)$$

$$\frac{value\ n}{tl \cdot (m :: n) \Rightarrow n} \quad (19)$$

$$\frac{D\ \rhd I \quad free\ C\ I \quad [D|C]N \Rightarrow n}{[\textbf{on}\ D\ \textbf{do}\ C]N \Rightarrow n} \quad (20)$$

$$\frac{M \Rightarrow m \quad [m/x]([D|C]N) \Rightarrow n}{[x = M; D|C]N \Rightarrow n} \quad (21)$$

In the following, $[C]N \stackrel{\text{def}}{=} [\langle\rangle|C]N$.

$$\frac{M \Rightarrow m \quad [m/x]N \Rightarrow n}{[x := M]N \Rightarrow n} \quad (22)$$

$$\frac{[C]([D]M) \Rightarrow m}{[C; D]M \Rightarrow m} \quad (23)$$

$$\frac{M \Rightarrow true \quad [C]N \Rightarrow n}{[\textbf{if}\ M\ \textbf{then}\ C\ \textbf{else}\ D]N \Rightarrow n} \quad (24)$$

$$\frac{M \Rightarrow false \quad [D]N \Rightarrow n}{[\textbf{if}\ M\ \textbf{then}\ C\ \textbf{else}\ D]N \Rightarrow n} \quad (25)$$

$$\frac{\begin{array}{c}M \Rightarrow true\\ [C]([\textbf{while}\ M\ \textbf{do}\ C]N) \Rightarrow n\end{array}}{[\textbf{while}\ M\ \textbf{do}\ C]N \Rightarrow n} \quad (26)$$

$$\frac{M \Rightarrow false \quad N \Rightarrow n}{[\textbf{while}\ M\ \textbf{do}\ C]N \Rightarrow n} \quad (27)$$

$$\frac{N \Rightarrow n \quad [[n/x]_c C]M \Rightarrow m}{[\mathbf{begin\ new}\ x = N;\ C\ \mathbf{end}]M \Rightarrow m} \tag{28}$$

$$\frac{\begin{array}{c} (x' \Rightarrow n) \\ \vdots \\ \mathit{value}\ n \quad [\alpha_{c\ x'/x} C]M \Rightarrow \underline{\alpha}_{x'/x} m \end{array}}{[[n/x]_c C]M \Rightarrow m} \tag{29}$$
x' is a new Id

$$\frac{\mathit{value}\ n}{\leq \cdot n \Rightarrow \leq \cdot n} \tag{30}$$

$$\frac{\mathit{value}\ n}{(\leq \cdot 0) \cdot n \Rightarrow \mathit{true}} \tag{31}$$

$$\frac{\mathit{value}\ n}{(\leq \cdot (\mathit{succ} \cdot n)) \cdot 0 \Rightarrow \mathit{false}} \tag{32}$$

$$\frac{(\leq \cdot n) \cdot m \Rightarrow p}{(\leq \cdot (\mathit{succ} \cdot n)) \cdot (\mathit{succ} \cdot m) \Rightarrow p} \tag{33}$$

$$\frac{[[\mathbf{lambda}\, x, y.C/P]_{pc} D]M \Rightarrow m}{[\mathbf{proc}\ P(x, y) = C\ \mathbf{in}\ D]M \Rightarrow m} \tag{34}$$

$$\frac{\begin{array}{c} (P' \Rightarrow_p \mathbf{lambda}\, x, y.C) \\ \vdots \\ \mathit{free}_c\ C\ (x, y) \quad [\alpha_{pc\ P'/P} D]M \Rightarrow \underline{\alpha}_{pe\ P'/P} m \end{array}}{[[\mathbf{lambda}\, x, y.C/P]_{pc} D]M \Rightarrow m} \tag{35}$$
P' is a new ProcId

$$\frac{\begin{array}{c} P \Rightarrow \mathbf{lambda}\, x, y.C \\ M \Rightarrow m \quad z \Rightarrow p \\ [p/x][m/y][C]x \Rightarrow v \quad [v/z]N \Rightarrow n \end{array}}{[P(z, M)]N \Rightarrow n} \tag{36}$$

$$\frac{\begin{array}{cc} \mathit{free}\ C\ (x, y) & \begin{array}{c} (\mathit{closed}\ P) \\ \vdots \end{array} \\ P \Rightarrow_p \mathbf{lambda}\, x, y.C & \mathbf{lambda}\, z.M \Rightarrow m \end{array}}{\mathbf{lambda}\, z.M \Rightarrow [\mathbf{lambda}\, x, y.C/P]_{pe} m} \tag{37}$$

$$\frac{\mathit{value}\ n \quad [Q/P]_{pe}(m \cdot n) \Rightarrow p}{([Q/P]_{pe} m) \cdot n \Rightarrow p} \tag{38}$$

$$\frac{\begin{array}{c} (P' \Rightarrow_p Q) \\ \vdots \\ \mathit{free}_c\ C\ (x, y) \quad \alpha_{pe\ P'/P} M \Rightarrow \underline{\alpha}_{pe\ P'/P} m \end{array}}{[\mathbf{lambda}\, x, y.C/P]_{pe} M \Rightarrow m} \tag{39}$$
P' is a new ProcId

A.1.4 Rules for judgment $\rhd$

$$\overline{\langle\rangle \rhd \mathit{nil}} \tag{40}$$

$$\frac{D_l \rhd I}{x = M; D_l \rhd x :: I} \tag{41}$$

A.1.5 Rules for judgment *closed*

$$\overline{\mathit{closed}\ m} \tag{42}$$
m is a constant

$$\frac{\mathit{closed}\ M \quad \mathit{closed}\ N}{\mathit{closed}(MN)} \tag{43}$$

$$\frac{\mathit{closed}\ m \quad \mathit{closed}\ n}{\mathit{closed}(m \cdot n)} \tag{44}$$

$$\frac{\begin{array}{cc} & \begin{array}{c} (\mathit{closed}\ x) \\ \vdots \end{array} \\ \mathit{closed}\ N & \mathit{closed}\ M \end{array}}{\mathit{closed}(\mathbf{let}\ x = N\ \mathbf{in}\ M)} \tag{45}$$

$$\frac{\begin{array}{cc} (\mathit{closed}\ f), (\mathit{closed}\ x) & (\mathit{closed}\ f) \\ \vdots & \vdots \\ \mathit{closed}\ N & \mathit{closed}\ M \end{array}}{\mathit{closed}(\mathbf{letrec}\, f(x) = N\ \mathbf{in}\ M)} \tag{46}$$

$$\frac{\begin{array}{c} (\mathit{closed}\ x) \\ \vdots \\ \mathit{closed}\ M \end{array}}{\mathit{closed}(\mathbf{lambda}\, x.M)} \tag{47}$$

$$\frac{\begin{array}{cc} & \begin{array}{c} (\mathit{closed}\ x) \\ \vdots \end{array} \\ \mathit{closed}\ n & \mathit{closed}\ M \end{array}}{\mathit{closed}([n/x]M)} \tag{48}$$

$$\frac{\mathit{closed}\ m \quad \mathit{closed}\ n}{\mathit{closed}(m :: n)} \tag{49}$$

$$\frac{D \rhd I \quad \mathit{free}\ C\ I \quad \mathit{closed}\ [D|\mathbf{nop}]M}{\mathit{closed}\ [\mathbf{on}\ D\ \mathbf{do}\ C]M} \tag{50}$$

$$\frac{\mathit{closed}\ M}{\mathit{closed}\ [\langle\rangle|\mathbf{nop}]M} \tag{51}$$

$$\frac{\begin{array}{cc} & \begin{array}{c} (\mathit{closed}\ x) \\ \vdots \end{array} \\ \mathit{closed}\ N & \mathit{closed}\ [R|\mathbf{nop}]M \end{array}}{\mathit{closed}\ [x = N; R|\mathbf{nop}]M} \tag{52}$$

$$\dfrac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y) \qquad \mathit{closed}\ M\end{array}}{\mathit{closed}\ [\mathbf{lambda}\, x,y.C/P]_{pe}M} \tag{53}$$

A.1.6 Rules for judgment *free*

$$\dfrac{}{\mathit{free}\ x\ (x,m)} \tag{54}$$

$$\dfrac{\mathit{free}\ x\ m}{\mathit{free}\ x\ (y,m)} \tag{55}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ m}{\mathit{free}\ (M\ N)\ m} \tag{56}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ x :: m}{\mathit{free}\ (\mathbf{let}\ x = M\ \mathbf{in}\ N)\ m} \tag{57}$$

$$\dfrac{\mathit{free}\ M\ x :: m}{\mathit{free}\ (\mathbf{lambda}\, x.M)\ m} \tag{58}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ m}{\mathit{free}\ (M \cdot N)\ m} \tag{59}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ N\ m}{\mathit{free}\ (M :: N)\ m} \tag{60}$$

$$\dfrac{\mathit{free}\ n\ m \quad \mathit{free}\ M\ x :: m}{\mathit{free}\ ([n/x]M)\ m} \tag{61}$$

$$\dfrac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y,m) \qquad \mathit{free}\ M\ m\end{array}}{\mathit{free}\ ([\mathbf{lambda}\, x,y.C/P]M)\ m} \tag{62}$$

$$\dfrac{\mathit{free}\ C\ m \quad \mathit{free}\ M\ m}{\mathit{free}\ ([C]M)\ m} \tag{63}$$

$$\dfrac{\mathit{free}\ C\ m \quad \mathit{free}\ D\ m}{\mathit{free}\ (C;D)\ m} \tag{64}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ m \quad \mathit{free}\ D\ m}{\mathit{free}\ (\mathbf{if}\, M\ \mathbf{then}\ C\ \mathbf{else}\ D)\ m} \tag{65}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ m}{\mathit{free}\ (\mathbf{while}\ M\ \mathbf{do}\ C)\ m} \tag{66}$$

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ x :: m}{\mathit{free}\ (\mathbf{begin\ new}\ x = M; C\ \mathbf{end}\ )\ m} \tag{67}$$

$$\dfrac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y,m) \qquad \mathit{free}\ D\ m\end{array}}{\mathit{free}\ (\mathbf{proc}\ P(x,y) = C\ \mathbf{in}\ D)\ m} \tag{68}$$

$$\dfrac{\mathit{closed}_p(P) \quad \mathit{free}\ x\ m \quad \mathit{free}\ M\ m}{\mathit{free}\ (P(x,M))\ m} \tag{69}$$

$$\dfrac{\mathit{free}\ n\ m \quad \mathit{free}\ C\ x :: m}{\mathit{free}\ ([n/x]C)\ m} \tag{70}$$

$$\dfrac{\begin{array}{c}(\mathit{closed}_p\ P)\\ \vdots\\ \mathit{free}\ C\ (x,y,m) \qquad \mathit{free}\ D\ m\end{array}}{\mathit{free}\ ([\mathbf{lambda}\, x,y.C/P]D)\ m} \tag{71}$$

A.2 NOS of $\mathcal{L}_D$

A.2.1 Rules for judgment *value*

$$\dfrac{R \Rightarrow_d m}{\mathit{value}\ m} \tag{72}$$

A.2.2 Rules for judgment $\Rightarrow$

$$\dfrac{R \Rightarrow_d r \quad \{r\}M \Rightarrow m}{\mathbf{let}\ R\ \mathbf{in}\ M \Rightarrow m} \tag{73}$$

$$\dfrac{M \Rightarrow m}{\{\mathit{nil}\}M \Rightarrow m} \tag{74}$$

$$\dfrac{[n/x]M \Rightarrow m}{\{x \mapsto n\}M \Rightarrow m} \tag{75}$$

$$\dfrac{\{r\}(\{s\}M) \Rightarrow m}{\{r :: s\}M \Rightarrow m} \tag{76}$$

A.2.3 Rules for judgment $\Rightarrow_d$

$$\dfrac{\begin{array}{c}(x' \Rightarrow n)\\ \vdots\\ \mathit{value}\ n \quad \alpha_{d\ x'/x}R \Rightarrow_d \underline{\alpha}_{x'/x}m\end{array}}{[n/x]_d R \Rightarrow_d m} \tag{77}$$

x' is a new variable

$$\dfrac{M \Rightarrow m}{x = M \Rightarrow_d x \mapsto m} \tag{78}$$

$$\dfrac{R \Rightarrow_d r \quad S \Rightarrow_d s}{R\ \mathbf{and}\ S \Rightarrow_d r :: s} \tag{79}$$

$$\dfrac{R \Rightarrow_d r \quad \{r\}_d S \Rightarrow_d s}{R;S \Rightarrow_d r :: s} \tag{80}$$

$$\dfrac{R \Rightarrow_d r}{\{\mathit{nil}\}_d R \Rightarrow_d r} \tag{81}$$

$$\dfrac{[n/x]_d R \Rightarrow_d r}{\{x \mapsto n\}_d R \Rightarrow_d r} \tag{82}$$

$$\dfrac{\{r\}_d(\{s\}_d R) \Rightarrow_d r}{\{r :: s\}_d R \Rightarrow_d r} \tag{83}$$

A.2.4 Rules for judgment *closed*

$$\frac{closed\ M}{closed\ \{nil\}M} \quad (84)$$

$$\frac{closed\ [n/x]M}{closed\ \{x \mapsto n\}M} \quad (85)$$

$$\frac{closed\ \{r\}\{s\}M}{closed\ \{r :: s\}M} \quad (86)$$

$$\frac{closed\ m}{closed\ x \mapsto m} \quad (87)$$

$$\frac{R \gg m \quad closed\ \langle m\rangle M}{closed(\mathbf{let}\ R\ \mathbf{in}\ M)} \quad (88)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M\end{array}}{closed\ \langle x\rangle M} \quad (89)$$

$$\frac{closed\ \langle m\rangle(\langle n\rangle M)}{closed\ \langle m :: n\rangle M} \quad (90)$$

A.2.5 Rules for judgment $\gg$

$$\frac{}{\langle\rangle \gg nil} \quad (91)$$

$$\frac{R \gg m \quad S \gg n}{R\ \mathbf{and}\ S \gg m :: n} \quad (92)$$

$$\frac{closed\ M \quad \begin{array}{c}(closed\ x)\\ \vdots \\ R \gg n\end{array}}{x = M; R \gg x :: n} \quad (93)$$

A.3 NOS of $\mathcal{L}_{M_F}$

A.3.1 Rules for judgment *value*

$$\frac{}{value(\mathbf{sig}\ B_{sig})} \quad (94)$$

A.3.2 Rules for judgment $\Rightarrow$

$$\frac{}{\mathbf{struct}\ \ \mathbf{end}\ \Rightarrow nil} \quad (95)$$

$$\frac{M \Rightarrow m \quad [m/x]\mathbf{struct}\ B_{str} \Rightarrow l}{\mathbf{struct}\ x = M\ B_{str} \Rightarrow (x \mapsto m, l)} \quad (96)$$

$$\frac{M \Rightarrow m \quad N \Rightarrow t \quad proj\ m\ (t)\ n}{M : N \Rightarrow n} \quad (97)$$

$$\frac{u \Rightarrow m \quad (x \mapsto p)\ in\ m}{u.x \Rightarrow p} \quad (98)$$

$$\frac{u \Rightarrow l \quad \{l\}M \Rightarrow m}{\mathbf{open}\ u\ \mathbf{in}\ M \Rightarrow m} \quad (99)$$

$$\frac{\begin{array}{c}M \Rightarrow m \quad u \Rightarrow l\\ upd\ l\ x\ m\ l' \quad [u := l']N \Rightarrow n\end{array}}{[u.x := M]N \Rightarrow n} \quad (100)$$

A.3.3 Rules for judgment *closed*

$$\frac{}{closed\ \mathbf{sig}\ B_{sig}} \quad (101)$$

$$\frac{closed\ M \quad closed\ N}{closed\ M : N} \quad (102)$$

$$\frac{}{closed\ \mathbf{struct}\ \ \mathbf{end}} \quad (103)$$

$$\frac{\begin{array}{c}(closed\ x)\\ \vdots \\ closed\ M \quad closed(\mathbf{struct}\ B_{str})\end{array}}{closed(\mathbf{struct}\ x = M\ B_{str})} \quad (104)$$

$$\frac{closed\ u}{closed\ u.x} \quad (105)$$

$$\frac{closed\ u \quad closed\ M}{closed(\mathbf{open}\ u\ \mathbf{in}\ M)} \quad (106)$$

A.3.4 Rules for judgments *in*, *proj*, *upd*

$$\frac{}{m\ in\ (m :: l)} \quad (107)$$

$$\frac{m\ in\ l}{m\ in\ (p :: l)} \quad (108)$$

$$\frac{}{proj\ l\ (\mathbf{sig}\ \ \mathbf{end}\ )\ nil} \quad (109)$$

$$\frac{(x \mapsto m)\ in\ l \quad proj\ l\ (\mathbf{sig}\ B_{sig})\ l'}{proj\ l\ (\mathbf{sig}\ x\ B_{sig})\ (x \mapsto m, l')} \quad (110)$$

$$\frac{}{upd\ (x \mapsto n, l)\ x\ m\ (x \mapsto m, l)} \quad (111)$$

$$\frac{upd\ l\ x\ m\ l'}{upd\ (y \mapsto n, l)\ x\ m\ (y \mapsto n, l')} \quad (112)$$
$$x \neq y$$

A.4 NOS of $\mathcal{L}_{M_I}$

A.4.1 Rules for judgment $\Rightarrow$

$$\dfrac{\begin{array}{c}\left(\begin{array}{c} R' \Rightarrow_m (m, R') \\ (R', P) \Rightarrow_{mp} \boldsymbol{\lambda} x, y.C \\ (R', f) \Rightarrow_{mf} \boldsymbol{\lambda} x.N \end{array}\right) \\ \vdots \\ M \Rightarrow m \quad \mathit{free}\ C\ (x, y) \quad \mathit{free}\ N\ (x) \qquad [\alpha_{mc\ R'/R} D]N' \Rightarrow \underline{\alpha}_{me\ R'/R} n' \end{array}}{[\textbf{module}\, R\ \textbf{is}\ x = M;\ \textbf{proc}\, P(y) = C;\ \textbf{func}\, f = N\, \textbf{in}\ D]N' \Rightarrow n'} \quad R' \text{ is a new } \mathit{ModId} \tag{113}$$

$$\dfrac{R \Rightarrow_m (p, R') \qquad (R', P) \Rightarrow_{mp} \boldsymbol{\lambda} x, y.C \qquad M \Rightarrow m \qquad [p/x][m/y][C]x \Rightarrow p' \qquad [p'/R]_m N \Rightarrow n}{[R.P(M)]N \Rightarrow n} \tag{114}$$

$$\dfrac{\begin{array}{c}(T \Rightarrow_m (p, R')) \\ \vdots \\ \mathit{value}\ p \qquad R \Rightarrow_m (\_, R') \quad \alpha_{me\ T/R} N \Rightarrow \underline{\alpha}_{me\ T/R}\, n \end{array}}{[p/R]_m N \Rightarrow n} \quad T \text{ is a new } \mathit{ModId} \tag{115}$$

$$\dfrac{R \Rightarrow_m (p, R') \qquad (R', f) \Rightarrow_{mf} \textbf{lambda}\, x.M \qquad [p/x]M \Rightarrow m}{R.f \Rightarrow m} \tag{116}$$

A.4.2 Rules for judgment *closed*

$$\dfrac{\mathit{closed}\ T}{\mathit{closed}\ T.f} \tag{117}$$

A.4.3 Rules for judgment *free*

$$\dfrac{\mathit{free}\ M\ m \quad \mathit{free}\ C\ (x, y) \quad \mathit{free}\ N\ (x) \quad \mathit{free}\ D\ (R, m)}{\mathit{free}\ (\textbf{module}\, R\ \textbf{is}\ x = M;\ \textbf{proc}\, P(y) = C;\ \textbf{func}\, f = N\, \textbf{in}\ D)\ m} \tag{118}$$

$$\dfrac{\mathit{free}\ R\ m \quad \mathit{free}\ M\ m}{\mathit{free}\ R.P(M)} \tag{119}$$

$$\dfrac{\mathit{free}\ R\ m}{\mathit{free}\ R.f\ m} \tag{120}$$

$$\dfrac{\mathit{free}\ p\ m \quad \mathit{free}\ N\ (R, m)}{\mathit{free}\ [p/R]_m N\ m} \tag{121}$$

A.4.4 Rules for judgment $\gg$

$$\dfrac{\begin{array}{c}(\mathit{closed}\ T) \\ \vdots \\ R \gg I\end{array}}{[p/T]_m R \gg I} \tag{122}$$

B Denotational semantics

B.1 Denotational semantics of $\mathcal{L}_P$

B.1.1 Semantic domains

$$
\begin{array}{ll}
\mathbb{V} = (\mathbb{N} + \mathbb{T} + \mathbb{U} + \mathbb{P} + \mathbb{F})_\perp^\top & \mathbb{P} = \mathbb{V} \times \mathbb{V} \\
\mathbb{N} = \mathit{Nat} \text{ (the domain of natural numbers)} & \mathbb{F} = \mathbb{V} \to \mathbb{V} \\
\mathbb{T} = \mathit{Truth} \text{ (the domain of truth values)} & \mathbb{E} = ((\mathit{Id} \to \mathbb{V}) \times (\mathit{ProcId} \to \mathbb{Q}))^\top \\
\mathbb{U} = \mathit{Unit} \text{ (the one-element domain)} & \mathbb{Q} = (\mathit{Id} \to \mathbb{V} \to \mathbb{E} \to \mathbb{E})^\top
\end{array}
$$

B.1.2 Operators

$$
\begin{array}{lll}
\mathit{newenv} & = (\lambda x.\top, \lambda p.\top) & : \mathbb{E} \\
\mathit{update} & = \lambda x.\lambda n.\underline{\underline{\lambda}}(\rho_v, \rho_p).([x \mapsto n]\rho_v, \rho_p) & : \mathit{Id} \to \mathbb{V} \to \mathbb{E} \to \mathbb{E} \\
\mathit{access} & = \lambda x.\underline{\underline{\lambda}}(\rho_v, \rho_p).\rho_v(x) & : \mathit{Id} \to \mathbb{E} \to \mathbb{V} \\
\mathit{procupdate} & = \lambda p.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p).(\rho_v, [p \mapsto q]\rho_p) & : \mathit{ProcId} \to \mathbb{Q} \to \mathbb{E} \to \mathbb{E} \\
\mathit{procaccess} & = \lambda p.\underline{\underline{\lambda}}(\rho_v, \rho_p).\rho_p(p) & : \mathit{ProcId} \to \mathbb{E} \to \mathbb{Q} \\
\mathit{overlay} & = \underline{\underline{\lambda}}\rho_1.\underline{\underline{\lambda}}\rho_2.\lambda x.\mathbf{if}\ \mathrm{is}\top(\rho_2(x)) \to \rho_1(x) \ [\!]\ \rho_2(x) & : \mathbb{E} \to \mathbb{E} \to \mathbb{E}
\end{array}
$$

B.1.3 Semantic functions

$$
\begin{array}{ll}
\mathcal{E} : \mathit{Expr} \to \mathbb{E} \to \mathbb{V} & \mathcal{D} : \mathit{Declarations} \to \mathbb{E} \to \mathbb{E} \\
\mathcal{C} : \mathit{Commands} \to \mathbb{E} \to \mathbb{E} & \mathcal{Q} : \mathit{Procedures} \to \mathbb{E} \to \mathbb{Q}
\end{array}
$$

$$
\mathcal{E}[\![x]\!] = \underline{\underline{\lambda}}\rho.\mathit{access}[\![x]\!]\rho \qquad \mathcal{E}[\![0]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{N}(\mathit{zero}) \qquad \mathcal{E}[\![\mathit{nil}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{U}()
$$

$$
\mathcal{E}[\![\mathit{true}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{T}(\mathit{true}) \qquad \mathcal{E}[\![\mathit{false}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{T}(\mathit{false})
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathit{succ}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v.\mathbf{cases}\ v\ \mathbf{of} \\
\qquad\qquad\qquad \mathrm{is}\mathbb{N}(m) \to \mathrm{in}\mathbb{N}(\mathit{plus}\ m\ \mathit{one}) \\
\qquad\qquad\qquad [\!]\ \mathrm{is}\mathbb{U}() \to \top \\
\qquad\qquad\qquad [\!]\ \mathrm{is}\mathbb{P}(c) \to \top \\
\qquad\qquad\qquad [\!]\ \mathrm{is}\mathbb{F}(f) \to \top \\
\qquad\qquad \mathbf{end})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathit{plus}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v_1.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v_2.\mathbf{cases}\ v_1\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathrm{is}\mathbb{N}(m_1) \to \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{cases}\ v_2\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{is}\mathbb{N}(m_2) \to \mathrm{in}\mathbb{N}(\mathit{plus}\ m_1\ m_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad [\!]\ \top \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{end} \\
\qquad\qquad\qquad\qquad\qquad [\!]\ \top \\
\qquad\qquad\qquad\qquad \mathbf{end}))
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\mathit{hd}]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v.\mathbf{cases}\ v\ \mathbf{of} \\
\qquad\qquad\qquad \mathrm{is}\mathbb{P}(c) \to c{\downarrow}_1 \\
\qquad\qquad\qquad [\!]\ \top \\
\qquad\qquad \mathbf{end})
\end{array}
$$

$$\mathcal{E}[\![tl]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v.\mathbf{cases}\ v\ \mathbf{of}\ \mathrm{is}\mathbb{P}(c) \to c\!\downarrow_2\ [\!]\ \top\ \mathbf{end})$$

$$\mathcal{E}[\![\leq]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v_1.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v_2.\mathbf{cases}\ v_1\ \mathbf{of}\ \mathrm{is}\mathbb{N}(m_1) \to \mathbf{cases}\ v_2\ \mathbf{of}\ \mathrm{is}\mathbb{N}(m_2) \to \mathrm{in}\mathbb{T}(\mathit{islessorequal}\ m_1\ m_2)\ [\!]\ \top\ \mathbf{end}\ [\!]\ \top\ \mathbf{end}))$$

$$\mathcal{E}[\![\mathbf{let}\ x = M\ \mathbf{in}\ N]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![N]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)$$

$$\mathcal{E}[\![\mathbf{letrec}\ f(x) = M\ \mathbf{in}\ N]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ g = \mathit{fix}(\underline{\underline{\lambda}}g.\underline{\underline{\lambda}}v.\mathcal{E}[\![M]\!](\mathit{update}\ [\![f]\!]\ g\ \rho)\ \mathbf{in}\ \mathcal{E}[\![N]\!](\mathit{update}\ [\![f]\!]\ g\ \rho)$$

$$\mathcal{E}[\![\mathbf{lambda}\ x.M]\!] = \underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{F}(\underline{\underline{\lambda}}v.\mathcal{E}[\![M]\!](\mathit{update}\ [\![x]\!]\ v\ \rho))$$

$$\mathcal{E}[\![M\ N]\!] = \underline{\underline{\lambda}}\rho.\mathbf{cases}\ \mathcal{E}[\![M]\!]\rho\ \mathbf{of}\ \mathrm{is}\mathbb{F}(f) \to f(\mathcal{E}[\![N]\!]\rho)\ [\!]\ \top\ \mathbf{end}$$

$$\mathcal{E}[\![M :: N]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v_1 = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathbf{let}\ v_2 = \mathcal{E}[\![N]\!]\rho\ \mathbf{in}\ \mathrm{in}\mathbb{P}((v_1, v_2))$$

$$\mathcal{E}[\![[m/x]N]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![m]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![N]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)$$

$$\mathcal{E}[\![m \cdot n]\!] = \underline{\underline{\lambda}}\rho.\mathbf{cases}\ \mathcal{E}[\![m]\!]\ \mathbf{of}\ \mathrm{is}\mathbb{F}(f) \to f(\mathcal{E}[\![n]\!]\rho)\ [\!]\ \top\ \mathbf{end}$$

$$\mathcal{E}[\![[\mathbf{on}\ \bar{x} = \bar{M}\ \mathbf{do}\ C]N]\!] = \underline{\underline{\lambda}}\rho.\mathbf{if}\ (\mathit{maxfree}\ [\![C]\!](\bar{x})) \to \mathcal{C}[\![C]\!](\mathcal{D}[\![\bar{x} = \bar{M}]\!]\rho)[\!]\ \top$$

where $\mathit{maxfree} : \mathit{Commands} \to \mathit{Id}^* \to \mathbb{T}$; the meaning of "$\mathit{maxfree}\ [\![C]\!]\ s = \mathit{true}$" is simply "every free identifier of C is in s". $\mathit{maxfree}$ is trivially defined on the syntactic structure of commands; we omit its definition. $\mathcal{D}[\![\langle\rangle]\!] = \underline{\underline{\lambda}}\rho.\mathit{newenv}$

$$\mathcal{D}[\![x = M; R]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathbf{let}\ \tau = \mathcal{D}[\![R]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)\ \mathbf{in}\ \mathit{overlay}\ \tau\ (\mathit{update}\ [\![x]\!]\ v\ \mathit{newenv})$$

$$\mathcal{D}[\![[n/x]_d R]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![n]\!]\rho\ \mathbf{in}\ \mathcal{D}[\![R]\!](\mathit{update}\ [\![x]\!]\ v\ \rho)$$

$$\mathcal{C}[\![x := M]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ \mathit{update}\ [\![x]\!]\ v\ \rho$$

$$\mathcal{C}[\![C; D]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ \tau = \mathcal{C}[\![C]\!]\rho\ \mathbf{in}\ \mathcal{C}[\![D]\!]\tau = \mathcal{C}[\![D]\!] \circ \mathcal{C}[\![C]\!]$$

$$\mathcal{C}[\![\mathbf{if}\ M\ \mathbf{then}\ C\ \mathbf{else}\ D]\!] = \underline{\underline{\lambda}}\rho.\mathbf{cases}\ \mathcal{E}[\![M]\!]\rho\ \mathbf{of}\ \mathrm{is}\mathbb{T}(t) \to \mathbf{if}\ t \to \mathcal{C}[\![C]\!]\rho\ [\!]\ \mathcal{C}[\![D]\!]\rho\ [\!]\ \top\ \mathbf{end}$$

$$\mathcal{C}[\![\mathbf{while}\ M\ \mathbf{do}\ C]\!] = \mathit{fix}(F)$$

where $F : (\mathbb{E} \to \mathbb{E}) \to (\mathbb{E} \to \mathbb{E})$ $F = \underline{\lambda} f.\underline{\underline{\lambda}}\rho.\mathbf{cases}\,\mathcal{E}[\![M]\!]\rho\,\mathbf{of}$

$$\begin{array}{l} \quad \mathrm{is}\mathbb{T}(t) \to \mathbf{if}\ t \to f(\mathcal{C}[\![C]\!]\rho)\ [\!]\ \rho \\ \quad [\!]\ \top \\ \mathbf{end} \end{array}$$

$$\begin{aligned} \mathcal{C}[\![\mathbf{begin\ new}\ x = M;\ C\ \mathbf{end}]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{let}\ \rho' = \mathit{update}\ [\![x]\!]\ (\mathcal{E}[\![M]\!]\rho)\ \rho\ \mathbf{in} \\ & \quad \mathbf{let}\ \rho'' = \mathcal{C}[\![C]\!]\rho'\ \mathbf{in} \\ & \quad\quad \mathit{update}\ [\![x]\!]\ (\mathit{access}\ [\![x]\!]\ \rho)\ \rho'' \end{aligned}$$

$$\begin{aligned} \mathcal{C}[\![[n/x]_c C]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{let}\ \rho' = \mathit{update}\ [\![x]\!]\ (\mathcal{E}[\![n]\!]\rho)\ \rho\ \mathbf{in} \\ & \quad \mathbf{let}\ \rho'' = \mathcal{C}[\![C]\!]\rho'\ \mathbf{in} \\ & \quad\quad \mathit{update}\ [\![x]\!]\ (\mathit{access}\ [\![x]\!]\ \rho)\ \rho'' \end{aligned}$$

$$\begin{aligned} \mathcal{C}[\![\mathbf{proc}\ P(x,y) = C\ \mathbf{in}\ D]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{let}\ \rho' = \mathit{procupdate}\ [\![P]\!]\ (\mathcal{Q}[\![\mathbf{lambda}\ x,y.C]\!]\rho)\ \rho\ \mathbf{in} \\ & \quad \mathbf{let}\ \rho'' = \mathcal{C}[\![D]\!]\rho'\ \mathbf{in} \\ & \quad\quad \mathit{procupdate}\ [\![P]\!]\ (\mathit{procaccess}\ [\![P]\!]\ \rho)\ \rho'' \end{aligned}$$

$$\mathcal{C}[\![P(x,M)]\!] = \ \underline{\underline{\lambda}}\rho.\mathbf{let}\ v = \mathcal{E}[\![M]\!]\rho\ \mathbf{in}\ ((\mathit{procaccess}\ [\![P]\!]\ \rho)\ [\![x]\!]\ v\ \rho)$$

$$\begin{aligned} \mathcal{C}[\![[Q/P]_{pc} C]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{let}\ \rho' = \mathit{procupdate}\ [\![P]\!]\ \mathcal{Q}[\![Q]\!]\rho\ \rho\ \mathbf{in} \\ & \quad \mathbf{let}\ \rho'' = \mathcal{C}[\![C]\!]\rho'\ \mathbf{in} \\ & \quad\quad \mathit{procupdate}\ [\![P]\!]\ (\mathit{procaccess}\ [\![P]\!]\ \rho)\ \rho'' \end{aligned}$$

$$\begin{aligned} \mathcal{Q}[\![\mathbf{lambda}\ x,y.C]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{if}\ \mathit{maxfree}\ [\![C]\!]\ ([\![x]\!],[\![y]\!])\,\mathit{emptysign}))\to \\ & \quad \lambda i.\underline{\underline{\lambda}} v_y.\underline{\underline{\lambda}}\tau.\mathbf{let}\ v_x = (\mathit{access}\ i\ \tau)\ \mathbf{in} \\ & \quad\quad \mathbf{let}\ \rho' = \mathcal{C}[\![C]\!](\mathit{update}\ [\![y]\!]\ v_y\ (\mathit{update}\ [\![x]\!]\ v_x\ \rho))\ \mathbf{in} \\ & \quad\quad\quad \mathit{update}\ i\ (\mathit{access}\ [\![x]\!]\ \rho')\ \tau \\ & \quad [\!]\ \top \end{aligned}$$

$$\mathcal{E}[\![[Q/P]_{pe} M]\!] = \ \underline{\underline{\lambda}}\rho.\mathcal{E}[\![M]\!](\mathit{procupdate}\ [\![P]\!]\ \mathcal{Q}[\![Q]\!]\rho\ \rho)$$

B.2 Denotational semantics of $\mathcal{L}_D$

B.2.1 Semantic functions

$$\mathcal{O} : \mathit{SyntEnvir} \to \mathbb{E} \to \mathbb{E}$$

$$\mathcal{E}[\![\mathbf{let}\ R\ \mathbf{in}\ N]\!] = \ \underline{\underline{\lambda}}\rho.\mathcal{E}[\![N]\!](\mathit{overlay}\ (\mathcal{D}[\![R]\!]\rho)\ \rho)$$

$$\mathcal{E}[\![\{r\}M]\!] = \ \underline{\underline{\lambda}}\rho.\mathcal{E}[\![M]\!](\mathcal{O}[\![r]\!]\rho) = \mathcal{E}[\![M]\!] \circ \mathcal{O}[\![r]\!]$$

$$\mathcal{D}[\![R\ \mathbf{and}\ S]\!] = \ \underline{\underline{\lambda}}\rho.\mathit{overlay}\ (\mathcal{D}[\![S]\!]\rho)\ (\mathcal{D}[\![R]\!]\rho)$$

$$\mathcal{O}[\![x \mapsto n]\!] = \ \underline{\underline{\lambda}}\rho.\mathit{update}\ (\mathcal{E}[\![n]\!]\rho)\ \rho$$

$$\mathcal{O}[\![r :: s]\!] = \ \mathcal{O}[\![s]\!] \circ \mathcal{O}[\![r]\!]$$

B.3 Denotational semantics of $\mathcal{L}_{M_F}$

B.3.1 Semantic domains

$$\begin{array}{ll} \mathbb{V} = (\mathbb{N}+\mathbb{U}+\mathbb{P}+\mathbb{F}+\mathbb{B}+\mathbb{S})^{\top}_{\bot} & \mathbb{B} = \mathit{Id} \times \mathbb{V} \\ \mathbb{U} = \mathit{Unit} & \mathbb{S} = \mathit{Id}^* = \mathbb{ES} + \mathbb{CS} \\ \mathbb{P} = \mathbb{V} \times \mathbb{V} & \mathbb{ES} = \mathit{Unit} \\ \mathbb{F} = \mathbb{V} \to \mathbb{V} & \mathbb{CS} = \mathit{Id} \times \mathbb{S} \end{array}$$

B.3.2 Operators

$$
\begin{array}{lll}
\textit{emptystruct} = \mathrm{in}\mathbb{U}() & : \mathbb{V} \\
\textit{consstruct} \;\; = \lambda x.\underline{\underline{\lambda}} v.\underline{\underline{\lambda}} c.\mathrm{in}\mathbb{P}(\mathrm{in}\mathbb{B}([\![x]\!], v), c) & : \mathit{Id} \to \mathbb{V} \to \mathbb{V} \to \mathbb{V} \\
\textit{emptysign} \;\; = \mathrm{in}\mathbb{ES}() & : \mathbb{V} \\
\textit{conssign} \;\; = \lambda i.\lambda t.\mathrm{in}\mathbb{CS}((i, t)) & : \mathit{Id} \to \mathbb{S} \to \mathbb{S} \\
\textit{accessstruct} = \text{see below} & : \mathit{Id} \to \mathbb{V} \to \mathbb{V} \\
\textit{applystruct} \;\; = \text{see below} & : \mathbb{V} \to \mathbb{E} \to \mathbb{E} \\
\textit{projection} \;\; = \text{see below} & : \mathbb{V} \to \mathbb{S} \to \mathbb{V} \\
\textit{longupdate} \;\; = \text{see below} & : \mathit{LongId} \to \mathbb{V} \to \mathbb{E} \to \mathbb{E}
\end{array}
$$

$$
\begin{array}{l}
\textit{projection} = \;\underline{\underline{\lambda}} s.\lambda t.\textbf{cases } t \textbf{ of} \\
\qquad\qquad \mathrm{is}\mathbb{ES}() \to \mathrm{in}\mathbb{U}() \\
\qquad\qquad [\!] \; \mathrm{is}\mathbb{CS}(i, t') \to \\
\qquad\qquad\qquad \textbf{let } v = \textit{accessstruct } i \; s \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } s' = \textit{projection } s \; t' \textbf{ in } \textit{consstruct } i \; v \; s' \\
\qquad\quad \textbf{end}
\end{array}
$$

$$
\begin{array}{l}
\textit{accessstruct} = \;\lambda i.\underline{\underline{\lambda}} s.\textbf{cases } s \textbf{ of} \\
\qquad\qquad \mathrm{is}\mathbb{U}() \to \top \\
\qquad\qquad [\!] \; \mathrm{is}\mathbb{P}(b, s') \to \\
\qquad\qquad\qquad \textbf{cases } b \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathrm{is}\mathbb{B}(j, v) \to \textbf{if } (i = j) \to v \; [\!] \; \textit{accessstruct } i \; s' \\
\qquad\qquad\qquad\qquad [\!] \; \top \\
\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad [\!] \; \top \\
\qquad\quad \textbf{end}
\end{array}
$$

B.3.3 Semantic functions

$$\mathcal{E}[\![\textbf{struct end}\,]\!] = \;\underline{\underline{\lambda}}\rho.\textit{emptystruct}$$

$$\mathcal{E}[\![x \mapsto n]\!] = \;\underline{\underline{\lambda}}\rho.\textbf{let } v = \mathcal{E}[\![n]\!]\rho \textbf{ in } \mathrm{in}\mathbb{B}([\![x]\!], v)$$

$$
\begin{array}{l}
\mathcal{E}[\![\textbf{struct } x = M \;\; B_{str}]\!] = \;\underline{\underline{\lambda}}\rho.\textbf{let } v_1 = \mathcal{E}[\![M]\!]\rho \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } v_2 = \mathcal{E}[\![\textbf{struct } B_{str}]\!](\textit{update } [\![x]\!] \; v_1 \; \rho) \textbf{ in} \\
\qquad\qquad\qquad\qquad\quad \textit{consstruct } [\![x]\!] \; v_1 \; v_2
\end{array}
$$

$$
\begin{array}{l}
\textit{applystruct} = \;\underline{\underline{\lambda}} s.\underline{\underline{\lambda}}\rho.\textbf{cases } s \textbf{ of} \\
\qquad\qquad \mathrm{is}\mathbb{U}() \to \rho \\
\qquad\qquad [\!] \; \mathrm{is}\mathbb{P}(b, s') \to \\
\qquad\qquad\qquad \textbf{cases } b \textbf{ of} \\
\qquad\qquad\qquad\qquad \mathrm{is}\mathbb{B}(i, v) \to \textit{applystruct } s'(\textit{update } i \; v \; \rho) \\
\qquad\qquad\qquad\qquad [\!] \; \top \\
\qquad\qquad\qquad \textbf{end} \\
\qquad\qquad [\!] \; \top \\
\qquad\quad \textbf{end}
\end{array}
$$

$$\mathcal{E}[\![\textbf{sig end}\,]\!] = \;\underline{\underline{\lambda}}\rho.\mathrm{in}\mathbb{S}(\textit{emptysign})$$

$$\begin{aligned}\mathcal{E}[\![\textbf{sig}\ x\ B_{sig}]\!] = {} & \underline{\underline{\lambda}}\rho.\textbf{cases}\ \mathcal{E}[\![\textbf{sig}\ B_{sig}]\!]\rho\ \textbf{of}\\ & \qquad \text{is}\mathbb{S}(s) \rightarrow \text{in}\mathbb{S}(\mathit{conssig}\ [\![x]\!]\ s)\\ & \qquad [\!]\ \top\\ & \textbf{end}\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![M : N]\!] = {} & \underline{\underline{\lambda}}\rho.\textbf{let}\ s = \mathcal{E}[\![M]\!]\rho\ \textbf{in}\\ & \quad \textbf{cases}\ \mathcal{E}[\![N]\!]\rho\ \textbf{of}\\ & \qquad \text{is}\mathbb{S}(t) \rightarrow \mathit{projection}\ s\ t\\ & \qquad [\!]\ \top\\ & \quad \textbf{end}\end{aligned}$$

$$\mathcal{E}[\![\textbf{open}\ u\ \textbf{in}\ M]\!] = \underline{\underline{\lambda}}\rho.\textbf{let}\ s = \mathcal{E}[\![u]\!]\rho\ \textbf{in}\ \mathcal{E}[\![M]\!](\mathit{applystruct}\ s\ \rho)$$

$$\mathcal{E}[\![u.x]\!] = \underline{\underline{\lambda}}\rho.\mathit{accessstruct}\ [\![x]\!]\ (\mathcal{E}[\![u]\!]\rho)$$

B.4 Denotational semantics of $\mathcal{L}_{M_I}$

B.4.1 Semantic domains

$$\begin{aligned}\mathbb{E} &= (\mathbb{IM} \times \mathbb{PM} \times \mathbb{MM})^{\top} & \mathbb{MM} &= \mathit{ModId} \rightarrow \mathbb{M}\\ \mathbb{IM} &= \mathit{Id} \rightarrow \mathbb{V} & \mathbb{M} &= (\mathbb{V} \times \mathbb{Q}_M \times \mathbb{F}_M)^{\top}\\ \mathbb{PM} &= \mathit{ProcId} \rightarrow \mathbb{Q} & \mathbb{Q}_M &= \mathbb{V} \rightarrow \mathbb{V} \rightarrow \mathbb{V}\\ \mathbb{Q} &= (\mathit{Id} \rightarrow \mathbb{V} \rightarrow \mathbb{E} \rightarrow \mathbb{E})^{\top} & \mathbb{F}_M &= \mathbb{V} \rightarrow \mathbb{V} = \mathbb{F}\end{aligned}$$

B.4.2 Operators

$$\begin{aligned}\mathit{newenv} &= (\lambda x.\top, \lambda p.\top, \lambda r.\top) && : \mathbb{E}\\ \mathit{update} &= \lambda x.\lambda n.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).([x \mapsto n]\rho_v, \rho_p, \rho_m) && : \mathit{Id} \rightarrow \mathbb{V} \rightarrow \mathbb{E} \rightarrow \mathbb{E}\\ \mathit{access} &= \lambda x.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_v(x) && : \mathit{Id} \rightarrow \mathbb{E} \rightarrow \mathbb{V}\\ \mathit{procupdate} &= \lambda p.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).(\rho_v, [p \mapsto q]\rho_p, \rho_m) && : \mathit{ProcId} \rightarrow \mathbb{Q} \rightarrow \mathbb{E} \rightarrow \mathbb{E}\\ \mathit{procaccess} &= \lambda p.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_p(p) && : \mathit{ProcId} \rightarrow \mathbb{E} \rightarrow \mathbb{Q}\\ \mathit{modupdate} &= \lambda r.\lambda q.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).(\rho_v, \rho_p, [r \mapsto m]\rho_m) && : \mathit{ProcId} \rightarrow \mathbb{Q} \rightarrow \mathbb{E} \rightarrow \mathbb{E}\\ \mathit{modaccess} &= \lambda r.\underline{\underline{\lambda}}(\rho_v, \rho_p, \rho_m).\rho_m(r) && : \mathit{ProcId} \rightarrow \mathbb{E} \rightarrow \mathbb{Q}\end{aligned}$$

B.4.3 Semantic functions

$$\mathcal{Q}' : \mathbb{Q} \rightarrow \mathbb{E} \rightarrow \mathbb{V} \rightarrow \mathbb{V} \rightarrow \mathbb{V}$$

$$\begin{aligned}&\mathcal{C}[\![\textbf{module}\ R\ \textbf{is}\ x = M;\ \textbf{proc}\ P(y) = C;\ \textbf{func}\ f = N\ \textbf{in}\ D]\!] =\\ &\quad \underline{\underline{\lambda}}\rho.\textbf{if}\ \mathit{maxfree}\ [\![C]\!]\ (\mathit{conssign}\ [\![x]\!]\ (\mathit{conssign}\ [\![y]\!]\mathit{emptysign})) \rightarrow\\ &\qquad \textbf{if}\ \mathit{maxfree}\ [\![N]\!]\ (\mathit{conssign}\ [\![x]\!]\ \mathit{emptysign}) \rightarrow\\ &\qquad\quad \textbf{let}\ m = \mathcal{E}[\![M]\!]\rho\ \textbf{in}\\ &\qquad\qquad \textbf{let}\ q = \mathcal{Q}'[\![\textbf{lambda}\ x, y.C]\!]\rho\ \textbf{in}\\ &\qquad\qquad\quad \textbf{let}\ g = \underline{\underline{\lambda}}v_x.\mathcal{E}[\![N]\!](\mathit{update}\ [\![x]\!]\ v_x\ \rho)\ \textbf{in}\\ &\qquad\qquad\qquad \textbf{let}\ \rho' = \mathit{modupdate}\ [\![R]\!]\ (m, q, g)\ \rho\ \textbf{in}\\ &\qquad\qquad\qquad\quad \mathit{modupdate}\ [\![R]\!]\ (\mathit{modaccess}\ [\![R]\!]\ \rho)\ \mathcal{C}[\![D]\!]\rho'\\ &\qquad [\!]\ \top\\ &\quad [\!]\ \top\end{aligned}$$

$$\mathcal{Q}'[\![\mathbf{lambda}\, x, y.C]\!] = \underline{\underline{\lambda}}\rho.\underline{\underline{\lambda}}v_x.\underline{\underline{\lambda}}v_y.access\ [\![x]\!]\ \mathcal{C}[\![C]\!](update\ [\![y]\!]\ v_y\ (update\ [\![x]\!]\ v_x\ \rho))$$

$$\begin{aligned}\mathcal{C}[\![R.P(M)]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{let}\ (n, q, g) = modaccess\ [\![R]\!]\ \rho\ \mathbf{in}\\ & \quad \mathbf{let}\ n' = q\ n\ (\mathcal{E}[\![M]\!]\rho)\ \mathbf{in}\\ & \quad\quad modupdate\ [\![R]\!]\ (n', q, g)\ \rho\end{aligned}$$

$$\mathcal{E}[\![R.f]\!] = \underline{\underline{\lambda}}\rho.\mathbf{let}\ (n, q, g) = modaccess\ [\![R]\!]\ \rho\ \mathbf{in}\ (g\ n)$$

$$\begin{aligned}\mathcal{E}[\![[n/R]_m M]\!] = \ & \underline{\underline{\lambda}}\rho.\mathbf{let}\ (\_, q, g) = modaccess\ [\![R]\!]\ \rho\ \mathbf{in}\\ & \quad \mathbf{let}\ m' = \mathcal{E}[\![n]\!]\rho\ \mathbf{in}\ \mathcal{E}[\![M]\!](modupdate\ [\![R]\!]\ (m', q, g)\ \rho)\end{aligned}$$

Acknowledgments

Many of the ideas in this paper were first presented in [Mic92]. The author is most grateful to Furio Honsell for his invaluable and patient support and for directing his thesis work. We wish to acknowledge Thierry Coquand, Frank Pfenning and Randy Pollack for several useful discussions and for making accessible via `ftp` the LEGO and Elf systems.

References

[AHM87] Arnon Avron, Furio Honsell, and Ian A. Mason. Using Typed Lambda Calculus to implement formal systems on a machine. Technical Report ECS-LFCS-87-31, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, July 1987.

[AS85] Harold Abelson and Gerard Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, Massachusetts, 1985.

[Avr91] Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, January 1991.

[BH90] Rod Burstall and Furio Honsell. Operational semantics in a natural deduction setting. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 185–214, Cambridge, June 1990. Cambridge University Press.

[Des86] Joelle Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First ACM Conference on Logic in Computer Science.* The Association for Computing Machinery, 1986.

[Don77] James E. Donahue. Locations considered unnecessary. *Acta Informatica*, 8:221–242, July 1977.

[Gen69] Gerhard Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North Holland Publishing Company, 1969.

[Han88] John J. Hannan. Proof-theoretical methods for analysis of functional programs. Technical Report MS–CIS–89–07, Dep. of Computer and Information Science, University of Pennsylvania, Pennsylvania, December 1988.

[Har89] Robert Harper. Introduction to Standard ML. Technical Report ECS–LFCS–86–14, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, January 1989.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

[Kah87] Gilles Kahn. Natural Semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, number 247 in LNCS, pages 22–39. Springer-Verlag, 1987.

[LPT89] Zhaohui Luo, Robert Pollack, and Paul Taylor. *How to use LEGO (A Preliminary User's Manual)*. Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, October 1989.

[Mar84] Per Martin-Löf. On the meaning of the logical constants and the justification of the logical laws. Technical Report 2, Scuola di specializzazione in Logica Matematica. Dipartimento di Matematica, Università di Siena, Siena, Italy. 1984

[Mic92] Marino Miculan. Semantica operazionale strutturata ad ambienti distribuiti – teoria e sperimentazione. Undergraduate thesis, University of Udine, Udine, Italy, July 1992. In italian.

[Mor73] J. H. Morris, Jr. Types are not sets. In *Conference Record of ACM Symposium on Principles of Programming Languages*, pages 120–124, Boston, October 1973. The Association for Computing Machinery.

[MP91] Spiro Michaylov and Frank Pfenning. Natural Semantics and some of its Meta-Theory in Elf. Technical Report MPI–I–91–211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.

[Pfe89] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322. IEEE, June 1989. Also available as ERGO Report 89–067, School of Computer Science, Carnegie Mellon Univ., Pittsburgh.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. DAIMI FN-19, Computer Science Department, Århus University, Århus, Denmark, September 1981.

[Sch86] David A. Schmidt. *Denotational Semantics.* Allyn & Bacon, 1986.

The ALF proof editor

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

September 1993

Abstract

Alf is an interactive proof editor. It is based on the idea that to prove a mathematical theorem is to build a proof object for the theorem. The proof object is directly manipulated on the screen, different manipulations correspond to different steps in the proof. The language we use is Martin-Löf's monomorphic type theory. This is a small functional programming language with dependent types. The language is open in the sense that it is easy to introduce new inductively defined sets. A proof is represented as a mathematical object and a proposition is identified with the set of its proof objects.

Background

During the years we have learned that there is no such thing as "the logic of programming". Different kinds of programs require different kind of reasoning. Programs are manipulating different kinds of objects, and it would be very awkward to code these objects into a fixed set of objects. Objects have their own logic, it is for instance very different to reason about ongoing processes and fixed objects like natural numbers and lists. We also need a different kind of logic when we are interested in computational aspects of a program (like complexity and storage requirements). I don't think we will ever find *the* logic of programming.

The idea behind a logical framework is to have a flexible formal logic, in which it is possible to introduce new kinds of objects, including objects for proofs. The logical framework we are using is Martin-Löf's monomorphic type theory, which can be seen as a small functional language with dependent types. We express problems as types and solutions (proofs) as programs.

The fundamental notion of proof is a process which leads to a conviction of something to hold (an assertion, or equivalently, a judgement). You have a series of steps, in each step you make an assertion which holds because earlier assertions have been made. A proof object should be a mathematical object which represents this proof process. The proof object must be derivable from the proof process. But we need something more: If a proof object represents a proof then it must be possible to compute a proof process from the proof object. Here is a difference between the polymorphic type theory and the monomorphic theory. In the monomorphic theory the proof objects really represents a proof of its type.

The traditional way of using a computer for interactive proof checking is to formalize the proof process and then letting the computer check each step. The user types in commands in some imperative language and the effect of executing a command is to update some internal data base which represents assertions being made. We call this indirect editing.

To directly build something with a computer is to have an impression that the objects which are built (and changed) are directly manipulated on the screen using the keyboard and the mouse. It is like we have a hand (represented by the cursor) on the screen to select parts and to grasp for different tools which can manipulate the object. A change of the object is immediately shown on the screen.

To indirectly build something is to issue commands to the computer (either by typing or by pointing). The command is performed and nothing happens on the screen. It is like we cannot see the objects being built, instead we have to make experiments on it to see what we have. Direct manipulation is better. You have an explicit picture of the object which you want to build in front of you. And the object is manipulated by manipulation of the picture of it. Parts of the object can be pointed to, deleted, moved and changed in various ways.

There is another distinction which we have to make, the one between interactive and batchwise building. To build an object batchwise is to build the entire object first and then check that it is correct. Interactive building is to build an object piece by piece. You start with an incomplete object and then fill in some parts of it. Erroneous building steps are immediately discovered.

The idea we use is to use a proof object as a true representative of a proof. The process of proving the proposition A is represented by the process of building a proof object of A. There is a close connection between the individual steps in proving A and the steps to build a proof object of A. For instance the act of applying a rule is done by building an application of a constant, to assume that a proposition A holds is to make an abstraction of a variable of the type A and to refer to an assumption is to use the corresponding variable.

We are interested in an interactive direct proof checker. So if we represent the proof process by the process of building a proof object it must be possible to deal with *incomplete* proof objects, i.e. proof objects which represents incomplete proofs.

The proof editor we are using can be seen as an interactive structure-oriented editor for Martin-Löf's monomorphic type theory. An object which has been created by the editor is always meaningful (well typed), which means that the object really represents a proof. Before we explain how a partial proof is represented, we will explain how a complete proof is represented. To do this we need to explain Martin-Löf's monomorphic type theory.

The logical framework

There are four judgement forms in type theory:

- A *type*. We know that A is a type when we know what it means to be an object in A.
- $A = B$. Two types are equal when they have the same objects, so an object in A must be an object of B and conversely. Identical objects in A must also be identical in B and vice versa.
- $a \in A$. a is an object in A.
- $a = b \in A$. a and b are identical objects in A.

These judgements are decidable.

How to form types

The type structure is very simple, there are two ways of forming ground types and one way of forming function types. I will use the notation $b[x := a]$ for the expression obtained by substituting the expression a for all free occurrences of the variable x in the expression b.

- Set is a type. This is the type whose objects are (inductively defined) sets.

  Set formation

$$\mathsf{Set}\ \mathit{type}$$

- If $A \in \mathsf{Set}$, i.e. if A is a set, then $\mathit{El}(A)$ is a type. The objects in this type are the elements of the set A. I will write A instead of $\mathit{El}(A)$, since it will always be clear from the context whether we mean A as a set (i.e. as an object in Set) or as a type.

  El-formation

$$\frac{A \in \mathsf{Set}}{\mathit{El}(A)\ \mathit{type}}$$

- If A is a type and B is a family of types for $x \in A$ then $(x \in A)B$ is the type which contains functions from A to B as objects. All free occurrences of x in B become bound in $(x \in A)B$.

  Fun formation

$$\frac{A\ \mathit{type} \qquad B\ \mathit{type}\ \ [x \in A]}{(x \in A)B\ \mathit{type}}$$

  To know that an object c is in the type $(x \in A)B$ means that we know that when we apply it to an object a in A we get an object $c(a)$ in $B[x := a]$ and that we get identical objects in $B[x := a_1]$ when we apply it to identical objects a_1 and a_2 in A.

How to form objects in a type

Objects in a type are formed from constants and variables using application and abstraction. I already mentioned how to apply a function to an object:

Application

$$\frac{c \in (x \in A)B \qquad a \in A}{c(a) \in B[x := a]}$$

Functions can be formed by abstraction, if $b \in B$ under the assumption that $x \in A$ then $[x]b$ is an object in $(x \in A)B$. All free occurences of x in b become bound in $[x]b$.

Abstraction

$$\frac{b \in B \ \ [x \in A]}{[x]b \in (x \in A)B}$$

The abstraction is explained by the ordinary β-rule which defines what it means to apply an abstraction to an object in A.

β – rule

$$\frac{a \in A \qquad b \in B \ \ [x \in A]}{([x]b)(a) = b[x := a] \in B[x := a]}$$

The traditional η-, α- and ξ-rules can be justified, i.e. under obvious type-restrictions the following equalities hold if $b_1 = b_2$:

$$\begin{array}{rcll} [x](c(x)) & = & c & \eta \\ [x]b & = & [y](b[x := y]) & \alpha \\ [x]b_1 & = & [x]b_2 & \xi \end{array}$$

I will sometimes use the notation $(A)B$ or $A \rightarrow B$ when B does not contain any free occurrences of x. I will write $(x_1 \in A_1, \ldots, x_n \in A_n)B$ instead of $(x_1 \in A_1) \ldots (x_n \in A_n)B$ and $b(a_1, \ldots, a_n)$ instead of $b(a_1) \ldots (a_n)$ in order to increase the readability. Similarly, I will write $[x_1] \ldots [x_n]e$ as $[x_1, \ldots, x_n]e$.

An object is *saturated* if it is not a function, i.e. if its type is Set or $El(A)$, for $A \in \mathsf{Set}$. The *arity* of an object is the number of arguments it can be applied to in order for the result to be saturated. It is an important property that a well-typed object has a unique arity.

Definitions

Most of the generality and strength of the language comes from the possibilities of introducing new constants. It is in this way that we can introduce the usual mathematical objects like natural numbers, integers, functions, tuples etc. It is also possible to introduce more complicated inductive sets like sets for proof objects.

A distinction is made between primitive and defined constants. The value of a *primitive constant* is the constant itself. So the constant has only a type, it doesn't have a definition. It gets its meaning in other ways (outside the theory). Such a constant is also called a constructor. Examples of primitive constants are N, succ and $\mathsf{0}$, they can be introduced by the following declarations:

$$\begin{array}{rcl} \mathsf{N} & \in & \mathsf{Set} \\ \mathsf{succ} & \in & \mathsf{N} \rightarrow \mathsf{N} \\ \mathsf{0} & \in & \mathsf{N} \end{array}$$

A defined constant is defined in terms of other objects. When we apply a defined constant to all its arguments in an empty context, e.g. $c(e_1, \ldots, e_n)$, then we get an expression which is a definiendum, i.e. an expression which computes in one step to its definiens (which is a well-typed object).

A defined constant can either be explicitly or implicitly defined. We declare an *explicitly defined constant* c by giving an abbreviation for it:

$$c \equiv a \in A$$

The constant c is a definiendum in itself, not only when it is applied to its arguments. For instance we can make the following explicit definitions:

$$\begin{array}{rcl} 1 & \equiv & \mathsf{succ}(\mathsf{0})) \in \mathsf{N} \\ I_{\mathsf{N}} & \equiv & [x]x \in \mathsf{N} \rightarrow \mathsf{N} \\ I & \equiv & [A][x]x \in (A \in \mathsf{Set}, A)A \end{array}$$

The last example is the monomorphic identity function which when applied to an arbitrary set A yields the identity function on A.

It is easy to check whether an explicit definition is correct, you just check that the definiens is an object in the correct type.

We declare an *implicitly defined constant* by showing what definiens it has when we apply it to its arguments. This is done by pattern-matching and the definition is sometimes recursive. Whether this kind of definition is meaningful can in general only be checked outside the theory. We must be sure that all well-typed expressions of the form $c(e_1, \ldots, e_n)$ is a definiendum with a unique welltyped definiens. Here are some examples:

$$\begin{array}{rcl}
+ & \in & \mathsf{N} \to \mathsf{N} \to \mathsf{N} \\
+(0, y) & \equiv & y \\
+(\mathsf{succ}(x), y) & \equiv & \mathsf{succ}(+(x, y)) \\
\mathsf{natrec} & \in & \mathsf{N} \to (\mathsf{N} \to \mathsf{N} \to \mathsf{N}) \to \mathsf{N} \to \mathsf{N} \\
\mathsf{natrec}(d, e, 0) & \equiv & d \\
\mathsf{natrec}(d, e, \mathsf{succ}(a)) & \equiv & e(a, \mathsf{natrec}(d, e, a))
\end{array}$$

The last example is a specialized version of the recursion operator, a more general form will be given later.

The representation of proofs, theories, theorems, derived rules etc.

We are representing proofs as mathematical objects, the type of a proof object represents the proposition which is the conclusion of the proof. Variables are used as names of assumptions and constants are used as rules. To apply a rule to a number of subproofs is done by applying a constant to the corresponding subproof objects.

A theory is presented by a list of typings and definitions of constants. When we read the constant as a name of a rule, then a primitive constant is usually a formation or introduction rule, an implicitly defined constant is an elimination rule (with the contraction rule expressed as the step from the definiendum to the definiens) and finally, an explicitly defined constant is a lemma or derived rule. As an example of this, consider the definition of conjunction.

The formation rule for conjunction expresses that $A \,\&\, B$ is a proposition if A and B are propositions:

$$\frac{A \ \mathit{prop} \qquad B \ \mathit{prop}}{A \,\&\, B \ \mathit{prop}}$$

We express this by introducing the primitive constant & by the following typing:

$$\& \in (\mathsf{Set},\ \mathsf{Set})\mathsf{Set}$$

We use the type of sets to represent the type of propositions. A canonical proof of the problem $A \,\&\, B$ is on the form $\&\mathbf{I}(a, b)$, where a is a proof of A and b a proof of B. This reflects the explanation that a proof of $A \,\&\, B$ consists of a proof of A and a proof of B. If we are in a context where A and B are propositions we can define $A \,\&\, B$ by introducing the primitive constant

$$\&\mathbf{I} \in (A, B) A \,\&\, B$$

Another notation for this is:

$$\frac{A \qquad B}{A \,\&\, B}$$

This is the introduction rule for conjunction. If we are in an empty context we must declare the parameters A and B explicitly:

$$\&\mathbf{I} \in (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A,\ B)A \,\&\, B$$

The elimination rules for conjunction

$$\frac{A \,\&\, B}{A} \qquad \frac{A \,\&\, B}{B}$$

are expressed by introducing the implicitly defined constants $\&\mathbf{E}_l$ and $\&\mathbf{E}_l$ by the following declarations:

$$\begin{array}{rcl} \&\mathbf{E}_l & \in & (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A \,\&\, B)A \\ \&\mathbf{E}_r & \in & (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A \,\&\, B)B \\ \&\mathbf{E}_l(A, B, \&\mathbf{I}(a, b)) & = & a \\ \&\mathbf{E}_r(A, B, \&\mathbf{I}(a, b)) & = & b \end{array}$$

The two last rules are the contraction rules for & and these are essential for the correctness of the elimination rule. Since all proofs of $A \,\&\, B$ is equal to a proof on the form $\&\mathbf{I}(a, b)$, where a is a proof of A and b a proof of B, we know from the contraction rule that we get a proof of A if we apply $\&\mathbf{E}_r$ to an arbitrary proof of $A \,\&\, B$, and similarly for $\&\mathbf{E}_l$.

To summarize, we have the following declarations and definitions for conjunction.

$$\begin{array}{rcl} \& & \in & (\mathsf{Set},\ \mathsf{Set})\mathsf{Set} \\ \&\mathbf{I} & \in & (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A,\ B)A \,\&\, B \\ \&\mathbf{E}_l & \in & (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A \,\&\, B)A \\ \&\mathbf{E}_r & \in & (A \in \mathsf{Set},\ B \in \mathsf{Set},\ A \,\&\, B)B \\ \&\mathbf{E}_l(A, B, \&\mathbf{I}(a, b)) & = & a \\ \&\mathbf{E}_r(A, B, \&\mathbf{I}(a, b)) & = & b \end{array}$$

The appendix contains a formalization of Martin-Löf's monomorphic set theory.

Representation of incomplete objects

When we are proving a proposition A in a theory then we are building a proof object of type A in an environment consisting of a list of declaration of constants. This is presented on the screen by having two windows, a theory window containing declarations of constants and a scratch area containing objects being edited. The scratch area contains different kind of buffers to build types and objects. A type buffer is used to build a type. The objects which are being built in the scratch area are always correct relative to the current theory.

Editing objects

When we are making a top-down proof of a proposition A, then we try to reduce the problem A to some subproblems $B_1, \ldots, B_n$ by using a rule c which takes proofs of $B_1, \ldots, B_n$ to a proof of A. Then we continue by proving $B_1, \ldots, B_n$. For instance, we can reduce the problem A to the two problems $C \supset A$ and C by using modus ponens. In this way we can continue until we have only axioms and assumptions left. This process corresponds exactly to how we can build a mathematical object from the outside and in. Suppose that we want to build an object like

$$f(g_1(a_1, a_2), g_2(b)).$$

Then we start from its outer form to build $f(?_1, ?_2)$, where $?_1$ and $?_1$ are placeholders for not yet filled-in objects, then continue to fill in g_1 or g_2 etc. This means that the type or the problem to solve is constant while the solution to it is edited. It is an important property of the formal system that it is possible to compute the expected type of the placeholders. It is because of this that we can look at the editing operations as a way of decomposing a problem into subproblems.

Let's see what kind of structure we need to represent incomplete objects. We will first introduce placeholders $?_1, \ldots ?_n$ to be used for parts of the objects which are to be filled in. The expression

$$? \in A$$

expresses a state of an ongoing process of finding an object in the type A. We say that the expected type of ? is A. Objects are built up from variables and constants using application and abstraction. Therefore there are four ways of refining a placeholder:

- The placeholder is replaced by a constant c. This is correct if the type of c is equal to A.
- The placeholder is replaced by a variable x. This is correct if the type of x is equal to A and if the expression replacing the placeholder may depend on x.
- The placeholder is replaced by an abstraction $[x]?_1$. We must have that

  $$[x]?_1 \in A$$

  which holds if A is equal to a functional type $(y \in B)C$. The type of the variable x must be B and we must remember that $?_1$ may be substituted by an expression which may depend on the variable x. We say that the *local context* of $?_1$ contains the typing $x \in B$. So after this refinement we have that

  $$?_1 \in C[y := x]$$

  and $?_1$ has a local context which contains $x \in B$. This corresponds to making a new assumption, when we are constructing a proof. We reduce the general problem $(y \in B)C$ to the problem $C[y := x]$ under the assumption that $x \in B$. The assumed object x can be used to construct a solution to C, i.e. we may use the knowledge that we have a solution to the problem B when we are constructing a solution to the problem C.

  Notice that the placeholder will in general be replaced by an open term, this is a motivation for having open terms as first-class objects.

- Finally, the placeholder can be replaced by an application $c(?_1, \ldots ?_n)$ where c is a constant, or $x(?_1, \ldots ?_n)$, where x is a variable. In the case that we have a constant, we must have that $c(?_1, \ldots ?_n) \in A$, which holds if the type of the constant c is equal to $(x_1 \in A_1, \ldots, x_n \in A_n)B$ and $?_1 \in A_1, ?_2 \in A_2[x_1 :=?_1], \ldots, x_n \in A_n[x_1 :=?_1, \ldots, x_{n-1} :=?_{n-1}]$ and
$$B[x_1 :=?_1, \ldots, x_{n-1} :=?_{n-1}] = A$$
So, we have reduced the problem A to the subproblems $A_1, A_2[x_1 :=?_1], \ldots, A_n[x_1 :=?_1, \ldots, x_{n-1}]$ and further refinements must satisfy the constraint $B[x_1 :=?_1, \ldots, x_{n-1} :=?_{n-1}] = A$. The number n of new placeholders can be computed from the arity of the constant c and the expected arity of the placeholder.

As an example, if we start with $? \in A$ and A is not a function type and if we apply the constant c of type $(x \in B)C$, then the new term will be
$$c(?_1) \in A$$
where the new placeholder $?_1$ must have the type B (since all arguments to c must have that type) and furthermore the type of $c(?_1)$ must be equal to A, i.e. the following equality must hold:
$$C[x :=?_1] \equiv A.$$
As will be described later, these kind of constraints will in general be simplified by the system. To summarize, the editing step from $? \in A$ to $c(?_1) \in A$ is correct if $?_1 \in B$ and $C[x :=?_1] \equiv A$. This operation corresponds to applying a rule when we are constructing a proof. The rule c reduces the problem A to the problem B.

Editing types top-down

A type buffer is initialized with a placeholder and this can be refined in three ways (corresponding to the three ways of forming types). One way is to refine it to Set. Another way is to refine it to a function type $(x \in ?_1)?_2$ where $?_1$ and $?_2$ are new placeholders standing for types (and $?_2$ may contain occurrences of the variable x). In this case, the place holder $?_1$ must be replaced by a complete expression before we go on to edit the placeholder $?_2$. Finally a type placeholder can be refined to $C(?_1, \ldots, ?_n)$ where C is an n-ary set forming operation (i.e. a constant of type $(x_1 \in A_1, \ldots, x_n \in A_n)\mathsf{Set}$). In this case the new placeholders stand for objects and we can refine them using the commands for editing objects.

Bottom-up buffers

A bottom-up buffer is used to edit (hypothetical) proofs (programs) bottom up. As the proof-term is edited, the type of it is computed by the system. When we want to build a term like
$$f(g_1(a_1, a_2), g_2(b))$$
then we start to build a_1, a_2 or b and then continue with $g_1(a_1, a_2)$ or $g_2(b)$ etc.

There are different ways of building an application $c(a_1, \ldots, a_n)$ from its parts. You can either update one of the arguments or update the expression $c(?_1, \ldots, ?_n)$. In the first case, the buffer is initialized with one of the arguments and there are ways to edit it. So, given an object a the buffer is initialized to
$$a \in A$$

where A is the type of a. It is now possible to use the object a as the first argument in an application, i.e. by pointing to a constant c replacing a with $c(a, ?_2, \ldots, ?_n)$ where $?_2, \ldots, ?_n$ are new placeholders, the number of which is decided by the arity of c. The editor then computes the type of the new object (and reject the editing operation if there is no typing). Of course, the type of the object depends in general on the placeholders.

Another way of building this object is to first build the term $c(?_1, \ldots, ?_n)$ and then replace the first placeholder with a. So instead of updating the object a, we update the object $c(?_1, \ldots, ?_n)$. The disadvantage of this is that there will be more buffers left after the final expression have been created. But only experience will tell what is best.

The theory window

A theory is a list of declarations of constants. Each constant has a type and the defined constants also have a definition (which is a well typed object). A theory can be edited by moving constant declarations between the scratch area and the theory. It is also possible to include declarations from a file. Declarations from an included file may not be changed without invoking the editor on that file.

Editing defined constants

Suppose that we are going to build an object with name c of type A. A buffer is then initialized to

$$c \in ?_1$$

$$c = ?_2 \quad []$$

and we can fill in the type A for the first placeholder using the commands for editing types. When we edit the definition of c we can choose to edit the lefthand side or the righthand side. If we double click on the lefthand side then it will change to

$$c(x_1, \ldots, x_n) = ? \qquad [x_1 \in A_1, \ldots, x_n \in A_n]$$

if A is definitionally equal to $(x_1 \in A_1, \ldots, x_n \in A_n)B$. If we now double click on the variable x_i, and if the type of x_i is a set, then the lefthand side will expand to

$$c(x_1, \ldots, c_1(y_1, \ldots, y_m), x_{i+1}, \ldots, x_n) = ?_1$$

$$c(x_1, \ldots, c_k(z_1, \ldots, z_u), x_{i+1}, \ldots, x_n) = ?_k$$

where $c_1, \ldots, c_k$ are all the constructors for the type of x_i. In this way we can create a definition by pattern matching and the system will guarantee that the patterns are exhaustive and mutually distinct.

When editing the righthand side of a defining equation it is possible to use a case construct on the outer level of the expression. This replaces the current placeholder with an expression of the form

case a of
    $c_1(y_1, \ldots, y_m) => ?_1$
| $c_k(z_1, \ldots, z_u) => ?_k$

Acknowledgements

This is joint work with Thierry Coquand, Lena Magnusson and Johan Nordlander. The first version of Alf was designed by Thierry and myself, Thierry implemented the first proof engine and Lennart implemented the user interface. The current proof engine is implemented by Lena. Johan has programmed the user interface.

Appendix: Definition of Martin-Löf's theory of sets

I will use the symbol $\downarrow$ in front of arguments which are not printed. These arguments can almost always be filled in automatically.

Cartesian product of a family of sets

The elements in the set $\Pi(A, B)$ are functions which takes an argument x in A to an element in $B(x)$. The set is used to express universal quantification and implication.

$$
\begin{array}{rcl}
\Pi & \in & (A \in \mathsf{Set},\ (A)\mathsf{Set})\ \mathsf{Set} \\
\lambda & \in & (\downarrow A \in \mathsf{Set}, \downarrow B \in (A)\mathsf{Set}, (x \in A)B(x)) \\
 & & \Pi(A, B) \\
\mathsf{apply} & \in & (\downarrow A \in \mathsf{Set},\ \downarrow B \in (A)\mathsf{Set},\ \Pi(A, B),\ x \in A) \\
 & & B(x) \\
\mathsf{apply}(\lambda(b), a) & = & b(a)
\end{array}
$$

We will write $\Pi x \in A.C$ instead of $\Pi(A, [x]C)$. We get the ordinary function set by making the explicit definition

$$A \to B = \Pi(A, [x]B)) \in \mathsf{Set}\ [A \in \mathsf{Set}, B \in \mathsf{Set}]$$

As usual, the following definitions are made:

$$
\begin{array}{rcl}
\forall & \equiv & \Pi \\
\forall I & \equiv & \lambda \\
\forall E & \equiv & \Pi E \\
\supset & \equiv & \to \\
\supset I & \equiv & \lambda \\
MP & \equiv & \mathsf{apply}
\end{array}
$$

Disjoint union of a family of sets

The elements in the set $\Sigma(A, B)$ are pairs consisting of an element a in in A and an element in $B(a)$. The set is used to express existential quantification, the cartesian product between two sets and conjunction.

$$
\begin{array}{rcl}
\Sigma & \in & (A \in \mathsf{Set},\ (A)\mathsf{Set})\ \mathsf{Set} \\
\Sigma I & \in & (\downarrow A \in \mathsf{Set},\ \downarrow B \in (A)\mathsf{Set}, x \in A, B(x))
\end{array}
$$

$$
\begin{array}{rcl}
 & & \Sigma(A, B) \\
\Sigma E & \in & (\downarrow A \in \mathsf{Set}, \downarrow B \in (A)\mathsf{Set}, \downarrow C \in (\Sigma(A,\ B))\mathsf{Set}, \\
 & & (x \in A,\ y \in B(x))\, C(\Sigma I(x,\ y)), \\
 & & p \in \Sigma(A,\ B)) \\
 & & C(p) \\
\Sigma E(d, \Sigma I(a, b)) & = & d(a, b)
\end{array}
$$

The usual cartesian product is defined by

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

We also make the following definitions:

$$
\begin{array}{rcl}
\times I & \equiv & \Sigma I \\
\times E & \equiv & \Sigma E \\
\& & \equiv & \times \\
\& I & \equiv & \times I \\
\& E & \equiv & \times E \\
\exists & \equiv & \Sigma \\
\exists I & \equiv & \Sigma I \\
\exists E & \equiv & \Sigma E
\end{array}
$$

Equality sets

The set $\mathsf{Id}(A, a, b)$ is the least reflexive relation, it is used to express that the elements a and b in A are equal.

$$
\begin{array}{rcl}
\mathsf{Id} & \in & (A \in \mathsf{Set}, A, A)\, \mathsf{Set} \\
\mathsf{refl} & \in & (\downarrow A \in \mathsf{Set}, x \in A)\, \mathsf{Id}(A, x, x) \\
\mathsf{idpeel} & \in & (\downarrow A \in \mathsf{Set}, x \in A, y \in A, \downarrow C \in (x \in A, y \in A, \mathsf{Id}(A, x, y))\, \mathsf{Set}, \\
 & & (z \in A)\ C(z, z, \mathsf{id}(z)), \\
 & & p \in \mathsf{Id}(A, x, y)) \\
 & & C(x, y, p) \\
\mathsf{idpeel}(A, a, b, C, d, \mathsf{id}(A, a)) & = & d(a)
\end{array}
$$

Finite sets

We introduce the empty set and the one element set as examples of finite sets. The empty set is introduced by declaring the constants

$$
\begin{array}{rcl}
\bot & \in & \mathsf{Set} \\
\mathsf{case}_\bot & \in & ((C \in \bot)\mathsf{Set},\ p \in \bot)\, C(p)
\end{array}
$$

Notice that there is no definition of the non-primitive constant $\mathsf{case}_\bot$.

The one element set is introduced by declaring the constants

$$\begin{array}{rcl}
\top & \in & \mathsf{Set} \\
\mathsf{tt} & \in & \top \\
\mathsf{case}_\top & \in & (\downarrow C\in(\top)\mathsf{Set},\ C(\mathsf{tt}),\ p\in\top)\,C(p) \\
\mathsf{case}_\top(b,\ \mathsf{tt}) & = & b(\mathsf{tt})
\end{array}$$

Natural numbers

The set of natural numbers is introduced by declaring the constants

$$\begin{array}{rcl}
\mathsf{N} & \in & \mathsf{Set} \\
\mathsf{0} & \in & \mathsf{N} \\
\mathsf{succ} & \in & (\mathsf{N})\,\mathsf{N} \\
\mathsf{natrec} & \in & (\downarrow C\in(\mathsf{N})\,\mathsf{Set}, \\
 & & C(\mathsf{0}), \\
 & & (x\in\mathsf{N}, C(x))\ C(\mathsf{succ}(x)), \\
 & & p\in\mathsf{N}) \\
 & & C(p) \\
\mathsf{natrec}(d, e, \mathsf{0}) & = & d \\
\mathsf{natrec}(d, e, \mathsf{succ}(a)) & = & e(a, \mathsf{natrec}(d, e, a))
\end{array}$$

Lists

$$\begin{array}{rcl}
\mathsf{List} & \in & (\mathsf{Set})\,\mathsf{Set} \\
\mathsf{nil} & \in & (A\in\mathsf{Set})\,\mathsf{List}(A) \\
\mathsf{cons} & \in & (\downarrow A\in\mathsf{Set}, A, \mathsf{List}(A))\,\mathsf{List}(A) \\
\mathsf{listrec} & \in & (\downarrow A\in\mathsf{Set},\ \downarrow C\in(\mathsf{List}(A))\,\mathsf{Set}, \\
 & & C(\mathsf{nil}(A)), \\
 & & (x\in A,\ y\in\mathsf{List}(A), C(x))\ C(\mathsf{cons}(x, y)), \\
 & & p\in\mathsf{List}(A)) \\
 & & C(p) \\
\mathsf{listrec}(d, e, \mathsf{nil}) & = & d \\
\mathsf{listrec}(d, e, \mathsf{cons}(a, b)) & = & e(a, b, \mathsf{listrec}(d, e, b))
\end{array}$$

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. Pattern matching with dependent types. In *In the informal proceeding from the logical framework workshop at Båstad*, June 1992.

[3] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[4] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

Developing certified programs in the system Coq
The Program tactic

C. Parent *
LIP, URA CNRS 1398, ENS Lyon
46 Allée d'Italie, 69364 Lyon cedex 07, France
e-mail : parent@lip.ens-lyon.fr

October 26, 1993

Abstract

The system *Coq* is an environment for proof development based on the Calculus of Constructions extended by inductive definitions. Functional programs can be extracted from constructive proofs written in *Coq*. The extracted program and its corresponding proof are strongly related. The idea in this paper is to use this link to have another approach: to give a program and to generate automatically the proof from which it could be extracted. Moreover, we introduce a notion of annotated programs.

1 Introduction

The system *Coq* is a proof development environment based on the Calculus of Constructions with inductive definitions [PM93, DFH$^+$93]. It uses the Curry-Howard isomorphism [How80], more precisely the fact that one can identify the notion of proofs and programs and the notion of types and specifications. It follows Heyting's semantics of constructive proofs : a proof of $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ gives a method to transform an object i and a proof of $P(i)$ into an object o and a proof of $Q(i,o)$. Systems following this interpretation like *Coq* can be seen as programming languages. Indeed, in a system like *Coq*, a proof of a specification is developed and can be represented as a program corresponding to the method in the Heyting's sense. In fact, a proof contains a lot of redundant informations : there are informations about the way of calculating the result (i.e. the interesting part) and informations about the way of calculating the correctness proof (i.e. the uninteresting part). So, following this idea that informations need to be removed, programs can be extracted from proofs in *Coq*, using a notion of realizability [PM89a]. Realizability is an interpretation of the computational contents of intuitionistic proofs as programs satisfying a given specification. Such a program is called a *realization* of the specification. Realizability allows to eliminate non computational parts of proofs (to extract programs from proofs) and to certify extracted programs to be still correct with respect to the initial specification. Indeed, from proofs written in *Coq*, programs can be extracted [PM89b] into a typed functional language like ML. Some other systems like PX and NuPrl offer similar possibilities of extraction [HN88, Con86]. Both of them are using untyped

*This research was partly supported by ESPRIT Basic Research Action "Types for Proofs and Programs" and by Programme de Recherche Coordonnées and CNRS Groupement de Recherche "Programmation".

theories. More precisely, PX uses an untyped theory and, in NuPrl, the theory is typed but the extracted terms are untyped. PX uses a notion of proofs-as-programs which is not the Curry-Howard one but there is two different levels (0 for proofs and 1 for programs). Then, a process of extraction is defined using a special notion of realizability called the px-realizability. A difference between the realizability used for the *Coq* extraction and the px-realizability is that the px-realizers are allowed not to terminate. In NuPrl, there is no distinction between proofs and programs. But, a process of extraction can be expressed : redundant informations can be hidden using the fact that if a is of type $\{x : A|P\ x\}$ then a is as well of type A, but a consequence of this is that typing becomes undecidable.

A problem when extracting programs from proofs is that proofs are first developed and, then, programs are extracted. This is not the case of other methods [BM92, Pol92] where proofs and programs are developed hand-by-hand. The aim here is based on the idea that a proof is developed differently if one waits for a program or another (for instance, different proofs of a same specification lead to different sort algorithms). A program can then be considered as a skeleton of its proof containing exactly all its computational contents. The aim in this paper is to develop a program and then to try to generate automatically using the program the proof of its specification. In fact, a program can be supposed to be a realization of its specification and, using this information, its proof can be generated almost automatically. But, two types of propositions have two be distinguished : *specifications* which have computational contents and are typically existential formulas such as $\forall x.P(x) \Rightarrow \exists y.Q(x,y)$ describing the properties to prove; *logical assertions* which have no computational contents (for instance, the pre and postconditions P and Q). Specifications can be automatically proved using the program but logical assertions can be arbitrarily complex and one cannot hope to solve them mechanically. The final aim is so to solve specifications and leave logical assertions to the user representing logical properties the program has to verify. Then, it can be certified that the program is correct if these properties are verified.

The method to develop automatically the proof from the program will use the structure of the program (which are variable, constant, abstraction, application or recursion). Each structure will give a certain method of proof.

The paper is organized as follows. In a first part, an example is developed in the system *Coq* in order to illustrate the program development method and introduce what we would like to obtain. In a second part, we give the methodology for automating the proof development. Then, we discuss some optimizations and conclude.

2 An example of development in *Coq*

Let us consider the division algorithm as an example of a development in *Coq*. A division program would take two arguments a and b and give as outputs q and r such that they verify $a = b * q + r \ \wedge \ b > r$. But, a necessary condition is that $b > 0$, otherwise the condition on r cannot be satisfied. So, a specification of a division algorithm should be :

$$\forall b.b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \ \wedge \ b > r) \tag{1}$$

Every constructive proof of such a specification gives an algorithm by extraction (see [PMW92]). If one gives a program, one gets the existence of such an algorithm and one has a skeleton of a possible proof. This skeleton allows to do the computational parts of the proof (i.e. to solve specifications). If the left logical assertions can be solved (e.q. prove that loop invariants are preserved), then the initial program can be certified correct with respect to the initial specification.

A program (in ML) for our example could be :

```
let div b a = divrec a where rec divrec = function
        0 -> (0,0)
      | Sn -> let (q,r) = divrec n in
                if (Sr<b) then (q,Sr) else (Sq,0) ;;
```

Let us do a mathematical proof of our specification and see the link with the ML program.

One wants to prove $\forall b. b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b*q+r \ \wedge \ b > r)$. Given b, $b > 0$, one wants to prove $\forall a.\exists q.\exists r.(a = b*q+r \ \wedge \ b > r)$. Let us do an induction on a. First case : $a = 0$. Then, one needs to prove $\exists q.\exists r.(0 = b*q+r \ \wedge \ b > r)$. The values $q = 0$ and $r = 0$ are good candidates since $0 = b*0+0 \ \wedge \ b > 0$. Second case : one assumes $\exists q.\exists r.(n = b*q+r \ \wedge \ b > r)$ for a given n, and one wants to prove $\exists q.\exists r.(Sn = b*q+r \ \wedge \ b > r)$. Given q and r from the induction hypothesis, let us look for q' and r' such that $(Sn = b*q'+r' \ \wedge \ b > r')$. Two subcases : if $Sr < b$ then let us take $q' = q$ and $r' = Sr$. Then, one has to prove $(Sn = b*q+Sr \ \wedge \ b > r)$. But, $b > Sr$ by hypothesis and one knows by the induction hypothesis that $n = b*q+r$. So, this case is solved. If $Sr \geq b$ then let us take $q' = Sq$ and $r' = 0$. Then, one has to prove $(Sn = b*q+b \ \wedge \ b > 0)$. The second part of the conjunction is trivial. For the first one, one knows $b > r$ and $Sr \geq b$, so one can conclude $b = Sr$. And the second case is solved.

Remark that the structure of the proof is closely related to the structure of the program : induction on a, recursive call on n in the second case of the induction ....

Let us now see how this proof can be developed in *Coq* and how a program can be extracted. Then, the link between proofs and programs will appear once more. First, *Coq* allows the interactive development of proofs. One gives a specification as above (1) and then one can use predefined tactics to develop a proof. The reasoning follows a natural deduction style. There are introduction (`Intro`) and elimination (`Elim`) tactics and resolution tactics (`Apply` or `Exists`). All the steps of the mathematical proof can be expressed with these tactics : the introductions by `Intro`, the inductions by `Elim`, the introductions of the existential quantifiers by `Exists` ... The proof of our example can then be expressed only using `Intro`, `Elim` and `Exists`. Comments are expressed between `(*` and `*)`.

```
Intros b H a.            (* H : b>0 *)
Elim a.
Exists 0. Exists 0.      (* b>0 and 0=b*0+0 *)
Auto.
Intros n H0.             (* H0 : induction hypothesis *)
Elim H0.                 (* getting back q *)
Intros q H1.
Elim H1.                 (* getting back r *)
Intros r H2.
Elim (inf b (S r)).      (* deciding of the order of b and Sr *)
Intros Le.               (* Le : b<=Sr *)
Exists (S q). Exists 0.  (* subgoal easy to resolve : b>0 and Sn=b*(Sq)+0 *)
Intros Gt.               (* Gt : b>Sr *)
Exists q. Exists (S r).  (* subgoal easy to resolve : b>Sr and Sn=b*q+Sr *)
```

This proof is close to the mathematical one in the sense that one can retrieve the steps of introductions, inductions .... Moreover, if one extracts the computational part from this *Coq* proof, one gets the program above. Our aim is now to take this program and to retrieve the computational parts of its corresponding proof.

Let us explain first how proofs and programs are represented in *Coq*. Proofs are typed λ-terms marked with informations on their computational contents (i.e. if they are informative or logical). The process of extraction consists in forgetting from the proof term all the logical parts to obtain a program term, so a typed λ-term with only informative parts. The extraction function is a forgetful function. Our aim is to inverse this function to obtain a proof term from a program term. The extraction function is defined on the structure of the proof terms. Thus, we have to define a function (that we will call the automation function) on the structure of the program terms. Now we describe the strategy of this automation function.

3 Automation method

As we said before, the principle is to give a specification and a program and to prove it is correct with respect to the specification. The method consists in associating the program to the current specification.

Then, an automatic proof (step-by-step) consists in applying the good tactic, giving as a result a new specification (or more) which is associated to a good new program (or more), which has to be a subprogram(s) of the previous program. Our method deals with *partial programs*, associated to *partial specifications* and builds *partial proofs*.

We first need to check that the type of the program is indeed convertible to the extraction of its specification (this property is kept as an invariant by our method). Indeed, one wants the program p to be the extraction $\mathcal{E}(P)$ of a proof P of the specification S; but, from the condition $P : S$, one gets $p : \mathcal{E}(S)$. One has to keep in mind this important information that the type of the program has to be the extraction of the specification.

Now, we explain how it will be done by cases on the structure of programs : λ-abstraction, application, recursion, variable and constant. We describe some heuristics and explain more precisely why annotated programs are sometimes necessary.

3.1 Programs

Programs are typed λ-terms given in a ${F_\omega}^{Ind}$ form (we consider that they are in normal form without apparent redexes). Their structure can be a λ-abstraction, an application, a recursion, a variable or a constant. Note that constants have a particular state. They can be considered like variables. But, in fact, they are programs already proved and they hide the structure of this program. In some cases, this information can be needed. If, they are just considered like variables, the information they hide is lost. So, when it is useful, constants are expanded to retrieve the structure of the program they correspond to and to use this important information.

In order to explain the method, we need to define what we call *coarse programs*.

Definition 1 *A program is coarse with respect to a specification S if it is exactly a proof of the specification S.*

Such programs are said to coarsely resolve the specification. They are interesting because they represent exactly the proof of their specification. They contain all the information useful for the proof. With such programs, a complex research of the corresponding proof can be avoided since it is directly in the program.

3.1.1 λ-abstractions

Let us first consider a typical example what kind of situations could appear. Consider the previous division algorithm written in the *Coq* syntax[1]

```
[b:nat][a:nat]
  <nat*nat>Match a with
    (* O *) <nat,nat>(O,O)
    (* S *) [n:nat][H:nat*nat]
              <nat*nat>let (q,r:nat) = H in
                  <nat*nat> if (inf b (S r)) then
                                 <nat,nat>((S q),O)
                              else <nat,nat>(q,(S r)).
```

whose specification is $\forall b.b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \ \wedge \ b > r)$. The program is a λ-abstraction. This indicates to introduce the b. This case is very simple. One introduces the b and then generates a new specification $b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \ \wedge \ b > r)$ for the program `[a:nat]...`. Now, to mimic the program, one would like to introduce the symbol a. But this introduction cannot be performed as the specification has not the shape $\forall a.\exists q.\exists r.(a = b * q + r \ \wedge \ b > r)$ but the shape $b > 0 \Rightarrow \forall a.\exists q.\exists r.(a = b * q + r \ \wedge \ b > r)$. In such a situation, one has to introduce the non-computational hypothesis $(b > 0)$ before the computational variable a. More generaly, before introducing a computational variable (corresponding to a variable of the program), one has to introduce all the non-computational hypotheses that have no equivalent in the program.

More formally, as one knows the correspondence between the program p and the specification S, one knows that if the program is a λ-abstraction ($p \equiv [x : \mathcal{E}(I)]p'$) then the specification is a product ($S \equiv (\vec{y} : \vec{L})(x : I)S'$ with $\vec{L}$ representing a vector of logical terms and I an informative term), since the type of the program is $(x : \mathcal{E}(I))A$ and is convertible to the extracted type of S. One has to do introductions. As we saw on the example, the problem is that one cannot be sure to have to do only one introduction. Indeed, a λ-abstraction in a program ($[x : \mathcal{E}(I)]p'$) corresponds to a product in the specification ($(x : I)S'$), but an informative one. The specification can contain non-informative products ($(\vec{y} : \vec{L})\ldots$). So, the method consists in this case in doing as many introductions as they are non-informative products in the specification and then one last introduction. This last introduction is about the proof term which corresponds to the λ-abstraction in the program.

This way, all the logical introductions of the proof that cannot appear in the program and then the "real" informative introduction the program indicates us are done. The new program associated to the new specification is the previous program without the corresponding λ-abstraction.

3.1.2 Applications

This case is more complex than the previous one. Let us take an example on parametric lists. Suppose one wants to prove $\forall l.\forall n.\exists m.(n + (\mathit{length}\ l) = m)$. A trivial proof is to explicitly give

[1]The notation `[x:A]B` is for the λ-term $\lambda x : A.B$, and `<P>Match x with` is the case analysis

$(n + (length\ l)$ as a witness that $\forall l.\forall n.\exists m.(n + (length\ l) = m)$. But, one wants to give a realizer to keep constructivity. So, a corresponding algorithm written in *Coq* could be :

```
[l:list]<nat->nat>Match l with
            [n:nat]n
            [a:A][m:list][H:nat->nat][n:nat](H (S n))
```

Let us look at the last part of this program `(H (S n))` where `H` is the induction hypothesis. The corresponding goal to prove is $\exists m.(n + (length\ (cons\ a\ l)) = m)$. The specification of `H` is $\forall n.\exists m.(n + (length\ l) = m)$ since `H` is the induction hypothesis. So, the specification of `(H (S n))` is $\exists m.(Sn + (length\ l) = m)$. This resolves the goal since $(length\ (cons\ a\ l)) = S(length\ l)$. Let us call this lemma `Length_l`.

Let us now show how programs already proved can be used in other programs. Suppose one wants to prove $\forall l.\exists m.((length\ l) = m)$. One can use the previous program and the associated program can be `[l:list](Length_l l O)`. Then, the specification of `(Length_l l O)` is $\exists m.(0 + (length\ l) = m)$. And this trivially resolves the goal.

Let us describe the method that is used. Let us write the program $(c\ a_1 \dots a_n)$ where c is not an application. Then, c is either a variable, a constructor or a recursion. Consider first when c is a variable or a constructor. The head symbol of the corresponding proof term is the same variable[2]. The proof has the shape $(c\ b_1 \dots b_p)$. Let $(x_1 : B_1) \dots (x_p : B_p)B$ be the type of c. One wants to generate the proof terms $b_1 \dots b_p$ of type $B_1 \dots B_p$. B_i with non-computational contents are left to the user, the others are associated to their extracted terms $a_1 \dots a_n$. Then, one applies to them the same method recursively.

Coarse programs can here be used as an optimization. If the specification is exactly the type of the program (coarse programs), then non-computational terms will never appear in the previous method. The proof is exactly the extracted term. So, one can use this extracted term for the proof term.

When c is a recursion. Then, there is no unique solution for the corresponding proof term. So, one chooses the following heuristic : if c is a recursion $Rec_I(m, P, lf)$[3] then its corresponding proof term is a recursion. Retrieving the proof terms corresponding to I and P is not easy because there are many solutions. But, in fact, finding the value of I is not very difficult since one can use the type of m. P is definitely a problem. Let us take the terms $(Rec_I(m, \lambda x : L.P, lf)\ x)$ and $Rec_I(m, P, lf)$. They are both in an η-long form and equivalent in terms of programs if x is a logical argument. So it is impossible to decide from programs which predicate one needs to take at the level of proof. To avoid failure, one has an heuristic corresponding to the following inference rules :

$$\frac{Rec_I(m, P, lf) : (x : A)\ B \quad a : A}{(Rec_I(m, P, lf)\ a) : B[x/a]}$$

$$\frac{Rec_I(m, P, lf) : A \to B \quad a : A}{(Rec_I(m, P, lf)\ a) : B}$$

One applies a generalization. The generalization of *name*, if *name* is a term on which depends the specification, replace it by the same specification quantified by the variable *name*.

[2] If c is a constructor $Constr(i, ind)$ then its corresponding proof term is a constructor $Constr(i, Ind)$ and, with respect to the *Coq* representation of inductive types, the current goal is an inductive type : $(y_1 : B_1) \dots (y_l : B_l)(Ind\ t_1 \dots t_k)$. Then, *Ind* is known from the current goal.

[3] $Rec_I(m, P, lf)$ is a notation for the *Coq* term : <P>*Match m with lf*

So, we use the previous inference rules to obtain two subproblems : one for the head of the application corresponding to a generalized specification, another for the argument (obviously, more if they are more than one arguments), corresponding to the specification of the argument (trivially solved if the argument is a coarse program).

So, one comes back to a case of recursion (see 3.1.3).Note it is not trivial to retrieve the specification of a program. This motivates the introduction of annotations in programs (see 4).

In the foregoing, one explained a method to resolve application cases whatever the head symbol of the application is. But, the problem of possible non-computational introductions (like in 3.1.1) has not been considered. The head of the specification can contain non-computational products. Logical introductions have to be done since they have no correspondent in the program. Let us take the example of the division algorithm at the step `((Sq),0)`. The specification is $(b \leq (Sr)) \rightarrow \exists q.\exists r.(Sn = b * q + r) \wedge (b > r)$. It is clear that one wants first to introduce $(b \leq (Sr))$ and then to resolve the goal with `((Sq),0)`. Thus, do all the logical introductions need to be done or not ?

Two kinds of introductions can be distinguished, dependent and non dependent ones. One says dependent introductions for introductions depending on the head of the specification, non dependent for the others. Consider the case of a bounded predicate variable as head symbol of the specification corresponding to induction principles. In such a case, one do not want to introduce non-computational hypotheses depending on the predicate variable (dependent hypotheses) since they could change after the induction. So, if the head symbol of the specification of c is not bounded then all the logical introductions are performed, else (bounded predicate variables case) only the logical non dependent introductions. Indeed, considering for example proofs by induction, since a proof of $A \rightarrow P(n)$ (with $n \notin A$) is equivalent and harder than a proof of $P(n)$ in the context of A, one chooses the second one. The equivalence of the two propositions is obvious but not the fact that the first could be harder than the second. Let us take the first case. Then, the transformation of the goal can give $(A \rightarrow P(n)) \rightarrow (A \rightarrow Q(n))$. The same transformation gives $P(n) \rightarrow Q(n)$ in the second case. And, it is clear, that the first case is harder and even sometimes impossible. Having explained why the logical hypotheses have to be introduced, we explain why dependent hypotheses should not be introduced. The reason is to keep the link between the hypothesis and the conclusion. Indeed, if the hypothesis is put in the context, then it can no more be modified though the conclusion can, and the link can be lost. And, the most probable situation is that this link needs to be used in the proof.

Finally, one can remark the description of the method for the application can be applied for a variable (remind that a constant is considered like a variable but expanded in case of failure). Moreover, note the importance of retrieving the specification of a part of a program and the fact that, if it is not possible, then the use of annotations ise motivated.

3.1.3 Recursions

This case is very similar to the previous one. But, let us see on a very simple example what could happen.

Suppose one wants to prove $\forall n.\forall m.(n \leq m) \vee (n > m)$ with the following associated program[4]

[4]Note that the specification of a function which returns a boolean value is a disjunction (and not an existential).

```
[n:nat](<nat->bool>Match n with
                [m:nat] true
                [n':nat][H:nat->bool][m:nat]
                        (<bool> Match m with
                                false
                                [m':nat][H':bool](H m'))).
```

Suppose one introduces the n. Then, one has an induction on n. The result has to be two new subspecifications for each case of the induction (the basic case and the induction case) associated to two new subprograms corresponding to the different cases (i.e. the different constructors of the inductive type of the induction element). The two specifications will be :$\forall m.(0 \leq m) \vee (0 > m)$ and $\forall n'.(\forall m.(n' \leq m) \vee (n' > m)) \rightarrow (\forall m.(Sn' \leq m) \vee (Sn' > m))$ with the two associated programs : `[m:nat] true`

and

```
[n':nat][H:nat->bool][m:nat]
   (<bool>Match m with
            false
            [m':nat][H':bool](H m'))
```

So, if the program is $Rec_I(m, P, lf)$, one wants to eliminate the proof corresponding to the program m. If m is coarse (previous example) then it is trivial else one needs to retrieve the specification of m by the previous method in order to eliminate it.

But note here the problem of logical introductions. The heuristics are the following. If the specification depends on m, then only all the logical non dependent introductions are done (like for the application). Otherwise, all the logical introductions are done.

But, there is another problem which we can illustrate with the following example. Let us take one more time our example of division algorithm. Suppose we take the subprogram (`inf` is a boolean funtion deciding the order of two natural numbers) :

```
<nat*nat>if (inf b (S r)) then <nat,nat>((S q),O)
         else <nat,nat>(q,(S r))
```

The corresponding specification is $(b > r) \rightarrow (n = b * q + r) \rightarrow \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$. The program suggests to do a case analysis on `(inf b (S r))`. First, the previous heuristic clearly appears to be necessary : $(b > r)$ and $(n = b * q + r)$ are introduced as logical non dependent hypothesis. Second, the specification does not depend on `(inf b (S r))`. Then, the case analysis will generate the two following identical subgoals (since the link with `(inf b (S r))` is lost) :

$$\exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

$$\exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

But, these subgoals are not useful since the information about wether `(inf b (S r))` is true or not is lost. The subgoals one would like to generate would rather be :

$$((\mathit{inf}\ \ b\ \ (S\ \ r)) = \mathit{true}) \rightarrow \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

$$((\mathit{inf}\ \ b\ \ (S\ \ r)) = \mathit{false}) \rightarrow \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

That is to introduce a dependency in the specification if the specification does not depend on the term of induction. So, if S is the current specification and t the proof term corresponding to m, then S is modified into $(t = t) \to S$. This just introduces a dependency without modifying the specification and allows to obtain probably more useful subgoals. Indeed, the problem comes from the fact that one looks for an adequate generalization of the goal and there are many ways of doing it. This choice is so only a heuristic.

Finally, one obtains as many subgoals as constructors of m and the different programs corresponding to the different constructors of the type of m are the elements of lf. Note, one more time, the importance of retrieving the specification of a part of a program since, if the specification of m cannot be retrieved, then the specification cannot be generalized.

4 Adding annotations

We saw the importance of retrieving a specification. Because it cannot always be done automatically, one would like to help the system. One need to add informations in the program. In fact, one wants to annot the program with comments which can be interpreted by the system. Let us add a new syntax for programs : one can annot any part of a program with the syntax `(: a specification :)`. Between `(:` and `:)`, one gives the specification one likes the program to have. This forces the system to take this information as a specification. Note that these annotations are available in a context of programs, that is to say that annotations are informations on a part of a program but can use programs variables. But, one can want to have a context of logical variables. So, one has to introduce another new syntax for λ-abstractions on logical variables `[{x:L}]`.

Let us give an example. Suppose we take the example of the division algorithm and particularly the step taken in 3.1.3. The specification is $(b > r) \to (n = b * q + r) \to \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$ and the program is `if (Sr<b) then (q,Sr) else (Sq,0)`. In fact, in *Coq*, it is written `<nat*nat>if (inf b (Sr)) then <nat,nat>((Sq),O) else <nat,nat>(q,(Sr))` with the constant `inf` being the decidability of the ordering relation on natural numbers.

Suppose first `inf` is declared as a variable without any specification, i.e. it is just a boolean value. Then, the generated subgoals are :

$$(inf\ \ b\ \ (Sr)) = true) \to \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

$$(inf\ \ b\ \ (Sr) = false) \to \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

But, then, one has to prove that :

$(inf\ \ b\ \ (Sr)) = true) \to (b \leq (Sr))$

$(inf\ \ b\ \ (Sr)) = false) \to (b > (Sr))$

which is not easy if `inf` has no specification.

In a second case, if `inf` is declared as a program already specified by $\forall n.\forall m.(n \leq m) \vee (n > m)$, then the generated subgoals are :

$$(b \leq (Sr)) \to \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

$$(b > (Sr)) \to \exists q \exists r\ (Sn = b * q + r) \wedge (b > r)$$

which are directly usable.

So, if `inf` is just a boolean function, one can explicitly indicate in the program the specification one wants for it by giving an annotation to this part of the program :

```
<nat*nat>if (inf b (Sr)) (: {(le b (Sr))}+{(gt b (Sr))} :)
         then <nat,nat>((Sq),O) else <nat,nat>(q,(Sr))
```

Then, this will generate the same subgoals as in the second case plus one needed to verify that the annotated program is consistent with its annotation. This last subgoal is the following :

$$(b \leq (Sr)) \vee (b > (Sr))$$

associated to the program `(inf b (Sr))`.

Now that we give this example, it is clear that the interpretation of annotations is that the specification of an annotated term is the annotation itself. So, let us give the following definition.

Definition 2 *A program p is said to computationally solve a specification S if there exist logical assertions L such that, if the set L is proved, then there exists a proof T of S such that the program p is an extraction of T.*

Then one can state the following claim :

Claim 3 *Every sufficiently annotated program can be computationally solved using the automation method described above, assuming some reasonable conditions on the dependencies in the specifications.*

Indeed, annotations allow to give specifications which cannot be retrieved automatically. So, if a program is sufficiently annotated then it can be computationally solved.

5 Optimizations

The method described above has been added to the system *Coq* (see [DFH$^+$93]). The entire library of *Coq* examples has been tested [Par92]. But, programs are written in a ${F_\omega}^{Ind}$ form which is not always very practical. We would like to have a language closer to ML. We have so introduced some optimizations in the formulation of the input program.

5.1 Recursive programs

The basic notion of *Coq* induction follows a primitive recursive scheme (or structural induction). But usual programs use a general recursion. *Coq* defines a well founded induction principle, which can be realized by a recursive program :

$$\forall P\ \forall R\ (\mathit{wellfounded}\ R) \rightarrow (\forall x\ (\forall y\ (R\ y\ x) \rightarrow (P\ y)) \rightarrow (P\ x)) \rightarrow \forall a.(P\ a)$$

The first optimization introduces a notion of recursive programs (general induction). A new syntax allows to write directly general recursive programs and it is translated in the previous well founded induction principle. So one needs an ordering relation R on which the induction is based and that has to be explicitly given. Indeed, this ordering relation cannot be retrieved automatically from the program and is the base of the well-foundness of the induction. With this new syntax, one gives the ordering relation and uses directly general recursive programs.

The syntax is `<P>rec H (: order :)` for the ML `let rec H x = ... H y ...` with `P` the type of the result and `order` the ordering relation on which the well-founded induction is based.

Example : Euclidean division algorithm.

An Euclidean division algorithm can be expressed by the following program :

```
[b:nat](<nat*nat>rec div (: lt :)
            [a:nat](<nat*nat>if (inf b a) then
                        (<nat*nat>let (q,r:nat) = (div (minus a b)) in
                                  <nat,nat>((S q),r))
                             else <nat,nat>(O,a))).
```

if `lt` is the natural strict ordering relation on natural integers, `div` and `minus` the division and subtraction on natural integers and `inf` the decidability on the ordering relation on natural numbers.

5.2 Eliminations

This optimization is in fact the inversion of an optimization done during the extraction. Suppose you have an elimination in a recursive program. An optimization of the extraction method is the following : if an hypothesis appears in each different case of the elimination, the program is transformed by taking this hypothesis off from each case and placing it just before the elimination. It is the current way of writing programs. If the extracted program is `<A->B>Match n with [x:A]t1 ... [x:A]tm`, then the more natural optimized program has a different shape : `[x:A]<B>Match n with t1 ... tm`. So, our optimization is to consider that every hypothesis which is external regarding an elimination and used in this elimination has to be placed back into each case of the elimination. This optimization is important because, if it is not done, it can generate much harder proofs.

Let us take an example. Suppose one has an hypothesis x before an elimination on a variable n (like previously), the specification of x can depend on n. If x is not moved inside the elimination then there are many chances that the hypothesis will not be the one expected, because, probably, one wants a different hypothesis for each different cases of the elimination and the proof should be more complicated and even impossible.

5.3 Contraction of expressions

The purpose here is to allow programs to be given in a natural form.

Let us take the case of expressions like `if b then true else false`. These expressions are not natural in programs, that is to say that a programmer writes only `b`, but the proof has to be explicitly given like this to correspond to the good specifications. For example, if b is correct with respect to A, it can be correct with respect to another specification B. But, the proof C from which b is extracted is of type A, but not of type B. For example, if A is $n = m \vee n \neq m$ and B is $Sn = Sm \vee Sn \neq Sm$, the program b realizes A and B but proves A and does not prove B. But, the program `if b then true else false` can be extracted from a proof of B. One has then to prove that $n = m \rightarrow Sn = Sm$ and $n \neq m \rightarrow Sn \neq Sm$.

So, the optimization consists in writing natural programs without `if b then true else false` and transforming them when necessary. This is just explained on this simple example but can be generalized and, then, automatic transformations of programs are generated.

5.4 Singleton types

Let us take the example of the Ackerman function to explain the problem of singleton types.

The definition of the function is the following :

$$\begin{aligned} ack(0,n) &= n+1 \\ ack(n+1,0) &= ack(n,1) \\ ack(n+1,m+1) &= ack(n,ack(n+1,m)) \end{aligned}$$

To express this definition we use in fact a ternary predicate :

```
Inductive Definition Ack : nat->nat->nat->Prop =
     Ack0 : (n:nat)(Ack O n (S n))
   | AcknO : (n,p:nat)(Ack n (S O) p)->(Ack (S n) O p)
   | AckSS : (n,m,p,q:nat)(Ack (S n) m q)->(Ack n q p)->(Ack (S n) (S m) p).
```

Suppose now we want to prove that $\forall n.\forall m.\exists p.Ack(n,m,p)$. The type extracted from this expression is : $nat \to nat \to sig\ nat$[5]. Suppose we want to give the program corresponding to this proof. It will be (in a CAML form) :

```
let rec ack n m = match n with
        0    -> (fun m -> m+1)
      | n'+1 -> (match m with
                           0    -> ack n' 1
                         | m'+1 -> ack n' (ack (n'+1) m')) ;;
```

This program written in a F_ω form[6] is :

```
[n:nat](<nat->nat>Match n with
        [m:nat](S m)
        [y:nat][H:nat->nat][m:nat]
                (<nat>Match m with
                        (H (S O))
                        [m':nat][H':nat](H H')))
```

The type of these last programs is : $nat \to nat \to nat$.

It is clear this type is not the same as the one extracted from the specification. But, suppose one develops the proof by hand of this specification, then the proof term will be (the Li represent logical parts which are not interesting from the program point of view) :

```
[n:nat](<[n0:nat](m:nat){p:nat|(Ack n0 m p)}>Match n with
[m:nat](exist (x:nat)([p:nat](Ack O m p) x) (S m) L1)
[y:nat][H:(m:nat){p:nat|(Ack y m p)}][m:nat]
  (<[n0:nat]{p:nat|(Ack (S y) n0 p)}>Match m with
       <[s:{p:nat|(Ack y (S O) p)}]{p:nat|(Ack (S y) O p)}>let
          (x:nat,p:(Ack y (S O) x)) = (H (S O)) in
              (exist (x0:nat)([p0:nat](Ack (S y) O p0) x0) x L2)
       [m':nat][H':sig nat]<[s:{p:nat|(Ack (S y) y0 p)}]{p:nat|(Ack (S y) (S y0) p)}>let
          (x:nat,p:(Ack (S y) y0 x)) = H' in
            <[s:{p0:nat|(Ack y x p0)}]{p0:nat|(Ack (S y) (S y0) p0)}>let
               (x':nat,p':(Ack y x x0)) = (H x) in
                  (exist (x1:nat)([p1:nat](Ack (S y) (S y0) p1) x1) x' L3)))
```

[5] $sig\ nat$ is the notation for the singleton type corresponding to nat, that is to say the inductive type with one constructor of type $nat \to (sig\ nat)$.

[6] The notation $[x:A]t$ denotes the λ-term $\lambda x:A.t$

with *exist* being the constructor of the inductive type *sig*. This proof has the same type as the specification. But, if one uses an isomorphism between A and $sig\ A$, one would like the proof and the program to have the same structure. And, this is not the case, since there are eliminations in the proof corresponding to the extraction of the structure of some terms of type $sig\ nat$ which do not appear in the natural program (since there are only terms of type nat). So, there are many transformations to do on the program to obtain a program able to generate the proof.

We define now a new notion of convertibility, a convertibility modulo $A \equiv sig\ A$ (that we will call **weak convertibility**). The sense is larger than the usual convertibility (that we will call **strong convertibility**) and allows to accept programs that have just to be modified. The method of transformation is based on a comparison of the program and its specification.

A first typical case is when the program is a λ-abstraction whose type is $A \to B$ when its specification type is $sig\ A \to C$[7]. The program is then transformed in a new λ-abstraction of type $sig\ A \to B$. For our example, let us take `H` of type $nat \to nat$ but the corresponding specification $\forall m.\exists p.Ack(y, m, p)$ is of type $nat \to sig\ nat$. So, the program is transformed with `[H:nat->(sig nat)]`. This implies that parts of programs will be no more well typed and this fact will help us to transform programs.

Another case of transformation is when the program is an application. There are two cases :

1. the program is ill typed. This implies that arguments are not of the good type A but of type $sig\ A$ and have to be replaced. For our example, let take `(H H')`. `H'` is of type $sig\ nat$ and `H` of type $nat \to sig\ nat$. This term is ill typed. So, it is transformed into :

   ```
   <sig nat>let (x:nat) = H' in (exist nat (H x)).
   ```

   We see the transformation is not complete. This is because of another problem. If we look at the specification of `(H x)` which is $\exists p.Ack(y + 1, m', p)$ and at the specification it is associated to which is $\exists p.Ack(y + 1, m' + 1, p)$, we see there are not identical. The program is then one more time transformed into :

   ```
   <sig nat>let (x:nat) = H' in <sig nat>let (x':nat) = (H x) in (exist nat x').
   ```

   This is in fact analogous to the transformation described in 5.3.

2. the program is well typed but of type A when its specification is of type $sig\ A$. For our example, let us take `(S m)`. It has type nat when its specification has type $sig\ nat$. The program is transformed into `(exist nat (S m))`.

This gives some typical cases of a method to transform programs which are weakly convertible but not strongly convertible to the specification. This allows to write programs in a more convivial form.

6 Comparaison with other works

As we said in the introduction, this method can be compared to the approach of [Pol92] and to the deliverables of [BM92]. [Pol92] describes a development of proofs and programs hand by hand. There is a separation of the programming language and of the logic language, which are two versions of the Calculus of Constructions. So, this is close to our approach but different in the sense that we first give the program and then develop automatically the proof. Moreover,

[7] C because it is B modulo $A \equiv sig\ A$.

there is a possibility of annotating programs to represent properties of these programs. With the deliverables approach, proofs and programs are too developed hand by hand. But, there is no separation between the programming language and the logical language. Proofs and programs are developed together using strong sums in the Luo's Extented Calculus of Constructions [Luo90]. This is what are called deliverables. There is a distinction between two kinds of deliverables : first-order ones which do not allow to express a relation between the input and the output, and second-order ones, which allow the expression of such a relation. Moreover, deliverables are more rigid than our approach in the sense that one cannot consider specifications not of the form $\forall x.(P\ x).\exists y.(Q\ x\ y)$.

7 Conclusion

The method presented above allows to obtain a system in which one can write programs and prove them automatically to be correct with respect to a specification. In fact, this method is not completely automatic since one usually has to solve logical assertions on the program by hand. Moreover, one has to comment programs with annotations, not in all cases but often. This allows to guide the proof but is not always trivial. One should have a more natural way of writing annotated programs, for example a possibility to suppress the type information in the programs and to replace it by annotations. Moreover, the future versions of *Coq* with existential variables [Dow91, Dowar] would allow to delay the instantiation of some parameters (like the ordering relation in the recursive programs) which could be fixed by the user when he solves the logical lemmas. Moreover, one could increase the synthesis power by using unification.

References

[BM92] R. Burstall and J. McKinna. Deliverables : a categorical approach to program development in type theory. Technical Report 92-242, LFCS, October 1992. Also in [NPP92].

[Con86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.

[DFH$^+$93] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq Proof Assistant User's Guide - Version 5.8. Technical Report 154, Projet Formel - INRIA-Rocquencourt-CNRS-ENS Lyon, May 1993.

[Dow91] G. Dowek. *Démonstration Automatique dans le Calcul des Constructions.* PhD thesis, Université Paris 7, 1991.

[Dowar] G. Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal of Logic and Computation*, To appear.

[HN88] S. Hayashi and H. Nakano. *PX : A Computational Logic.* Foundations of Computing. MIT Press, 1988.

[How80] W.A. Howard. The formulaes-as-types notion of construction. In J.R. Hindley, editor, *To H.B.Curry : Essays on Combinatory Logic , lambda-calculus and formalism.* Seldin, J.P., 1980.

[Luo90] Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, Department of Computer Science, University of Edinburgh, June 1990.

[NPP92] B. Nordström, K. Petersson, and G. Plotkin, editors. *Prooceedings of the 1992 worshop on types for proofs and programs*, June 1992.

[Par92] C. Parent. Automatisation partielle du développement de programmes dans le système Coq. Master's thesis, Ecole Normale Supérieure de Lyon, June 1992.

[PM89a] 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.

[PM89b] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* PhD thesis, Université Paris VII, 1989.

[PM93] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, March 1993. Also in research report 92-49, LIP-ENS Lyon, December 1992.

[PMW92] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation-special issue on automated programing*, 1992. To appear.

[Pol92] E. Poll. A programming logic for Fω. Technical Report 92/25, Eindhoven University of Technology, September 1992.

Refinement Types for Logical Frameworks

Frank Pfenning
Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3891, U.S.A.

Internet: fp@cs.cmu.edu

July 1993

Abstract

We propose a refinement of the type theory underlying the LF logical framework by a form of subtypes and intersection types. This refinement preserves desirable features of LF, such as decidability of type-checking, and at the same time considerably simplifies the representations of many deductive systems. A subtheory can be applied directly to hereditary Harrop formulas which form the basis of λProlog and Isabelle.

1 Introduction

Over the past two years we have carried out extensive experiments in the application of the LF Logical Framework [HHP93] to represent and implement deductive systems and their metatheory. Such systems arise naturally in the study of logic and the theory of programming languages. For example, we have formalized the operational semantics and type system of Mini-ML and implemented a proof of type preservation [MP91] and the correctness of a compiler to a variant of the Categorical Abstract Machine [HP92]. LF is based on a predicative type theory with dependent types. It has proved to be an excellent language for such formalization efforts, since it allows direct representation of deductions as objects and judgments as types and supports common concepts such as variable binding, substitution, and generic and hypothetical judgments. The logic programming language Elf [Pfe91a] implements LF and gives it an operational interpretation so that LF signatures can be executed as logic programs. It also provides sophisticated term reconstruction, which is important for realistic applications.

Despite its expressive power, certain weaknesses of LF emerged during these experiments. One of these is the absence of any direct form of subtyping. Clearly, this is not a theoretical problem: what is informally presented as subtyping can be encoded either via explicit coercions or via auxiliary judgments as we will illustrate below. In practice, however, this becomes a significant burden, and encodings are further removed from informal mathematical practice than desirable.

An obvious candidate for an extension of the type system are *subset types* as they are used for example in Martin-Löf type theory [SS88]. In a logical framework, however, they are problematic, because they lead to an undecidable type-checking problem. The methodology of LF reduces proof checking in the object language to type checking in the meta-language (the LF type theory), and thus decidability is important. Looking elsewhere, we find an extensive body of work on *order-sorted* first-order calculi and their use in logic programming and automated theorem proving (see, for example, [Smo89, SS89]). However, it is not clear how to generalize these calculi to logics or type theories with higher-order functions, although recently some interesting work in this direction has begun [Koh92, NQ92]. Similar systems of simple subtypes have been used in programming languages, in particular in connection with record types and object-oriented programming, but such systems are not expressive enough for our purposes. More promising are enhancements of simple subtypes with *intersection types* [CDCV81], which have been applied to programming languages [Rey91] and recently also in type theory [Hay91]. General decidability of type-checking or inference in such calculi is problematic, but under certain restrictions type checking is decidable and principal types exist [Rey88, FP91, CG92].

In this paper we tie together ideas from these threads of research and propose a refinement of the LF type theory by a version of bounded intersection types, or *refinement types*, as we call them. The resulting type theory $\lambda^{\Pi\&}$ allows more direct encodings of deductive systems in many examples. We show that it has a decidable type-checking problem and is thus useful as a logical framework. We have not yet implemented this system, but experience with a related implementation of refinement types for ML [FP91] and the current Elf term reconstruction algorithm leads us to believe that type-checking will be practical. While similar in spirit to the work on refinement types for ML [FP91], the technical and practical issues in both systems are very different. In ML, we are concerned with the decidability of *type inference* in the presence of general recursion and polymorphism. Here, we have to deal with *type checking* in a language without recursion or polymorphism, but with dependent types. Furthermore, in ML refinement types are defined inductively; here refinement types are open-ended in the same way that signatures are essentially open-ended (they can be extended with further declarations without invalidating earlier declarations).

The system we propose is relevant not only to LF and its Elf implementation, but a restricted version can be applied directly to λProlog [MNPS91] and Isabelle [PN90] with similar benefits. A unification algorithm for this restricted λ-calculus, $\lambda^{\rightarrow\&}$ is described in [KP93].

In future work, we plan to consider the operational aspects of this type theory so that it can be fully embedded into the current Elf implementation. This includes extending the constraint solving algorithm in [KP93] to account for dependencies in the style of [Pfe91a, Pfe91b], type reconstruction, and search. Based on experience from first-order logic programming we conjecture that subtyping constraints can lead to improved operational behavior of many programs.

2 Two Motivating Examples

In this section we give two prototypical examples which motivate our extension of the LF type theory. Space only permits a rather sketchy discussion of these examples; the interested reader may find additional explanation in the indicated references.

Hereditary Harrop Formulas. Here we consider, as an object logic, the language of hereditary Harrop formulas [MNPS91], a fragment of logic suitable as a basis for a logic programming language. For the sake of brevity we restrict ourselves to the propositional formulas.

$$\textit{Formulas} \quad F \quad ::= \quad A \mid F_1 \wedge F_2 \mid F_1 \supset F_2 \mid F_1 \vee F_2$$

Here A ranges over atomic formulas. We now define legal program and goal formulas.

$$\begin{array}{llll} \textit{Programs} & D & ::= & A \mid D_1 \wedge D_2 \mid G \supset D \\ \textit{Goals} & G & ::= & A \mid G_1 \wedge G_2 \mid G_1 \vee G_2 \mid D \supset G \end{array}$$

How do we represent these definitions in LF? The definition of formulas given here in the concrete syntax of Elf, is straightforward.

```
form : type.

=>   : form -> form -> form.  %infix right 10 =>
||   : form -> form -> form.  %infix right 12 ||
&&   : form -> form -> form.  %infix right 14 &&
```

Atomic formulas are not explicitly declared, but we assume that declarations for predicate constants are added to this basic signature as they are introduced. The next question is how to represent programs and goals. Here we can go two ways: one is to introduce explicit judgments *atom* F, *prog* F, and *goal* F which can be used to prove that a given formula F is either an atom, program, or goal. That is, showing only the rules for programs:

```
atom : form -> type.
goal : form -> type.
prog : form -> type.

p_atom : atom A -> prog A.
p_imp  : goal A -> prog B -> prog (A => B).
p_and  : prog A -> prog B -> prog (A && B).
```

Here, free variables in a declaration are implicitly Π-quantified.

A judgment, such as $P \vdash G$ (program P entails goal G) must now carry explicit evidence that the constituents P and G are in fact legal programs and goals. We call this judgment *solve* P G, indicating its use as a logic program. It requires *backchain* as an auxiliary judgment. `{x:A} K` is Elf's concrete syntax for $\Pi x{:}A.\ K$.

```
solve     : {P:form} prog P -> {G:form} goal G -> type.
backchain : {P:form} prog P -> {A:form} atom A -> {G:form} goal G -> type.
```

The rules defining these judgments lead to a very awkward and inefficient implementation of proof search, since *solve* is now a type family indexed by four arguments instead of only two.

Another possibility is to declare separate types for programs and goals. Unfortunately, this means that we have to introduce separate instances of the shared connectives, and the connection to an overarching language of formulas is lost and would have to be axiomatized separately.

Both alternatives illustrate general techniques available within the LF type theory. While feasible for relatively small examples, they become very difficult to manage for larger examples and obscure the representations greatly compared to the relative simplicity of the informal definition. In contrast, with refinement types we can declare a type of formulas and then atoms, programs, and goals as subtypes.

Natural Deductions in Normal Form. The next example illustrates that we often want to make subtype distinctions at the level of deductions and not only at the level of syntax. We follow the usual representation of natural deduction in LF [HHP93] and Felty's trick to enforce normal forms [Fel89]. We restrict ourselves to the purely implicational fragment.

$$\dfrac{\begin{array}{c}\overline{A}^{\,x}\\ \vdots\\ B\end{array}}{A \supset B}\supset\mathrm{I}^x \qquad\qquad \dfrac{A \supset B \qquad A}{B}\supset\mathrm{E}$$

The deduction in the premise of the implication introduction rule discharges the hypothesis A labelled x and is represented as a function from deductions of A to deductions of B. The derivability judgment is represented by the family *pf* which is indexed by a formula.

```
o   : type.
imp : o -> o -> o.

pf  : o -> type.

impi : (pf A -> pf B) -> pf (imp A B).
impe : pf (imp A B) -> pf A -> pf B.
```

Again, quantifiers over A and B are implicit. A type of the form *pf* A is the type of all natural deductions of A. A natural deduction is *normal* if no introduction of an implication is immediately followed by its elimination. An equivalent formulation essentially says that we can only reason with elimination rules from hypotheses and with introduction rules from the conclusion. We implement this via two judgments, *elim* and *nf*, on deductions. This has the same drawbacks as in the previous example: it is more verbose, and arguments proliferate in judgments which depend on *elim* and *nf*. Here is how this alternative could be written:

```
nf   : pf A -> type.
elim : pf A -> type.

impi_nf   : {Q:pf A -> pf B} ({P:pf A} elim P -> nf (Q P)) -> nf (impi Q).
impe_elim : {P:pf (imp A B)} {Q:pf A} elim P -> nf Q -> elim (impe P Q).
elim_nf   : {P:pf A} elim P -> nf P.
```

Implicit arguments (to *nf*, *elim*, *impi*, and *impe*) and type reconstruction in Elf go a long way towards making this option feasible, but it is still awkward. Felty's solution introduces new families *elim* and *nf* indexed by formulas. Again, the connection to *pf* remains informal and one then has to prove that every normal natural deduction is in fact a natural deduction. Using refinement types, we will be able to declare deductions in normal form as a subtype of natural deductions.

3 The Refinement Type System

In this section we present a refinement of the LF type theory (λ^{Π}) to accomodate commonly used forms of subtypes. We refer to this system as $\lambda^{\Pi\&}$. We have to ensure that the basic, necessary properties of the LF type theory are not destroyed: in particular, we need to preserve decidability of type-checking and the adequacy of encodings. These requirements have led us to a number of basic design decision which we review here before the technical development. The examples will draw upon Section 2.

Sorts and Proper Types. Semantically, a *sort* may be best thought of as describing a subset of a *proper type* as it exists in LF. This extends through the type hierarchy in straightforward fashion; for example, the sort $(\mathit{elim}\ A \to \mathit{nf}\ B)$ will describe a subset of the functions of type $(\mathit{pf}\ A \to \mathit{pf}\ B)$, namely those that map a deduction of A by elimination rules to a normal form deduction of B. Thus we think of sorts as a *refinement* of the structure of types, and similary for sort families indexed by objects. Sorts are not distinguished syntactically, but via a new form of declaration that specifies a sort refining a type. For example, $\mathit{goal} :: \mathit{form}$ declares the sort *goal* of legal goals as a refinement of the type *form* of formulas.

Subsorts and Intersection Types. The space of sorts that refine a given proper type must possess structure to be useful. We thus introduce new declarations of the form $a \leq a'$ that specify that sort a is a subsort of sort a'. This will only be considered well-formed when both a and a' refine some proper type b. At the level of functions, simple subsorting is insufficient, since a given λ-expression may have a number of different sorts. For example, $(\lambda x{:}\mathit{pf}\ A.\ x)$ has type $\mathit{pf}\ A \to \mathit{pf}\ A$, and also sorts $\mathit{elim}\ A \to \mathit{elim}\ A$ and $\mathit{nf}\ A \to \mathit{nf}\ A$. In order to express all these properties directly we use intersection types:

$$(\lambda x{:}\mathit{pf}\ A.\ x) : (\mathit{elim}\ A \to \mathit{elim}\ A) \,\&\, (\mathit{nf}\ A \to \mathit{nf}\ A) \,\&\, (\mathit{pf}\ A \to \mathit{pf}\ A).$$

Again, in keeping with the basic refinement philosophy, sorts may only be conjoined if they refine a common type ($\mathit{pf}\ A \to \mathit{pf}\ A$, in this example).

Objects. We also make a basic decision not to change the space of objects, but merely to classify them more accurately than in λ^{Π}. This may seem rather drastic insofar as types occur in objects (labelling λ's) and one might thus expect them to change as the language of types changes. Through the typing rules we enforce that λ-abstractions are labelled by proper types. The typing rules then allow analysis of the body of the term $\lambda x{:}A.\ M$ for every sort that refines the type A. This restriction may not be necessary to obtain a decidable system, but it affords a tremendous simplification of the meta-theory of our calculus without affecting its expressiveness in any essential way. It is also consistent with the philosphy behind refinement types.

3.1 Syntax

We maintain LF's three levels and augment families and kinds by intersections. Objects and contexts remain basically the same, although we have eliminated family-level abstractions $\lambda x{:}A_1.\ A_2$, since they do not occur in normal forms and are thus not important in practice.

$$\begin{array}{llcl}
\mathit{Kinds} & K & ::= & \mathrm{Type} \mid \Pi x{:}A.\ K \mid K_1 \,\&\, K_2 \\
\mathit{Families} & A & ::= & a \mid A\, M \mid \Pi x{:}A_1.\ A_2 \mid A_1 \,\&\, A_2 \\
\mathit{Objects} & M & ::= & c \mid x \mid \lambda x{:}A.\ M \mid M_1\, M_2 \\
\mathit{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}A
\end{array}$$

Signatures may now contain two additional forms of declarations: refinement declarations $a_1 :: a_2$ and subsort declarations $a_1 \leq a_2$.

$$\mathit{Signatures}\quad \Sigma \quad ::= \quad \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A \mid \Sigma, a_1 :: a_2 \mid \Sigma, a_1 \leq a_2$$

We now also drop the restriction that a constant may be declared at most once in a signature (where $a{:}K$, $a_1 :: a_2$, and $c{:}A$ declare a, a_1, and c, respectively). Instead we impose other validity conditions in the next section. As usual, we consider α-convertible terms to be identical.

3.2 Judgments

In our approach, it is extremely important that sorts and sort families can be recognized, and that a sort refines a unique type. Thus we begin by defining the refinement judgment. Since it must be applied uniformly through all levels (kinds, families, objects) with essentially the same rules, we use the meta-variables U and V to range over terms from any of the three levels and d to range over object-level or family-level constants. For an instance of a rule schema to be valid it must be sensible according to the stratification imposed above. Variables occurring in the terms involved in this judgment are treated uniformly, so we omit the context here.

$$\frac{}{\vdash_\Sigma \mathrm{Type} :: \mathrm{Type}} \qquad \frac{\vdash_\Sigma U_1 :: V_1 \qquad \vdash_\Sigma U_2 :: V_2}{\vdash_\Sigma \Pi x{:}U_1.\ U_2 :: \Pi x{:}V_1.\ V_2}$$

$$\frac{\vdash_\Sigma U_1 :: V_1 \qquad \vdash_\Sigma U_2 :: V_2}{\vdash_\Sigma U_1\, U_2 :: V_1\, V_2} \qquad \frac{\vdash_\Sigma U_1 :: V \qquad \vdash_\Sigma U_2 :: V}{\vdash_\Sigma U_1 \,\&\, U_2 :: V}$$

$$\frac{}{\vdash_\Sigma x :: x} \qquad \frac{\vdash_\Sigma U_1 :: V_1 \qquad \vdash_\Sigma U_2 :: V_2}{\vdash_\Sigma \lambda x{:}U_1.\ U_2 :: \lambda x{:}V_1.\ V_2}$$

$$\frac{d{:}U \text{ in } \Sigma}{\vdash_\Sigma d :: d} \qquad \frac{a :: a' \text{ in } \Sigma}{\vdash_\Sigma a :: a'}$$

Note that the refinement relation is neither transitive nor reflexive. The conditions on valid signatures will guarantee that exactly one of the last two cases is applicable for any declared constant, and the second only for a unique a'. This implies that in a valid signature Σ for a given U there exists at most one V such that $\vdash_\Sigma U :: V$.

The validity judgments have the following form. Here, Kind is a special token to allow a uniform presentation of the validity judgments at the three levels.

$\vdash \Sigma$ Sig	*Σ is a valid signature*
$\vdash_\Sigma \Gamma$ Ctx	*Γ is a valid context*
$\Gamma \vdash_\Sigma K$: Kind	*K is a valid kind*
$\Gamma \vdash_\Sigma A : K$	*A is a valid family of kind K*
$\Gamma \vdash_\Sigma M : A$	*M is a valid object of type A*

We also need the auxiliary judgments

$U \equiv V$	*U is $\beta\eta$-convertible to V*
$\vdash_\Sigma U \leq V$	*U is a subsort of V*

where the subsorting judgment only applies at the levels of families and kinds. Here are the rules for valid signatures.

$$\frac{}{\vdash \cdot\ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma\ \mathrm{Sig} \qquad \vdash_\Sigma K : \mathrm{Kind} \qquad \vdash_\Sigma K :: K' \qquad \vdash_\Sigma K_i :: K' \text{ for any } a{:}K_i \text{ in } \Sigma \qquad \text{no } a :: a' \text{ in } \Sigma}{\vdash \Sigma, a{:}K\ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma\ \mathrm{Sig} \qquad \vdash_\Sigma A : \mathrm{Type} \qquad \vdash_\Sigma A :: A' \qquad \vdash_\Sigma A_i :: A' \text{ for any } c{:}A_i \text{ in } \Sigma}{\vdash \Sigma, c{:}A\ \mathrm{Sig}}$$

$$\frac{\vdash \Sigma \text{ Sig} \qquad a_2{:}K \text{ in } \Sigma \qquad a_1 \text{ not declared in } \Sigma}{\vdash \Sigma, a_1 :: a_2 \text{ Sig}}$$

$$\frac{\vdash \Sigma \text{ Sig} \qquad a_1 :: a_3 \text{ in } \Sigma \qquad a_2 :: a_3 \text{ in } \Sigma}{\vdash \Sigma, a_1 \leq a_2 \text{ Sig}}$$

A declaration of the form $a :: b$ declares a *sort family* a which inherits its kind from the type family b it refines. Valid contexts are straightforward.

$$\frac{}{\vdash_\Sigma \cdot \text{ Ctx}} \qquad \frac{\vdash_\Sigma \Gamma \text{ Ctx} \qquad \Gamma \vdash_\Sigma A : \text{Type}}{\vdash_\Sigma \Gamma, x{:}A \text{ Ctx}}$$

The rules for valid terms are uniform throughout the levels (as long as they apply), so we give them in schematic form for terms. Note that we do not check validity of signatures or contexts at the leaves, but require their validity in the theorems and take care to propagate this property. Where there is no ambiguity we use the usual conventions for the names of meta-variables. Here, S stands for either Type or Kind.

$$\frac{}{\Gamma \vdash_\Sigma \text{Type} : \text{Kind}} \qquad \frac{x{:}A \text{ in } \Gamma}{\Gamma \vdash_\Sigma x : A}$$

$$\frac{d{:}U \text{ in } \Sigma}{\Gamma \vdash_\Sigma d : U} \qquad \frac{a :: b \text{ in } \Sigma \qquad b{:}K \text{ in } \Sigma}{\Gamma \vdash_\Sigma a : K}$$

$$\frac{\Gamma \vdash_\Sigma U : V_1 \qquad \Gamma \vdash_\Sigma U : V_2}{\Gamma \vdash_\Sigma U : V_1 \,\&\, V_2}(1) \qquad \frac{\Gamma \vdash_\Sigma U : V \qquad \vdash_\Sigma V \leq W \qquad \Gamma \vdash_\Sigma W : S}{\Gamma \vdash_\Sigma U : W}(2)$$

$$\frac{\Gamma \vdash_\Sigma A : \text{Type} \qquad \Gamma, x{:}A \vdash_\Sigma U : S}{\Gamma \vdash_\Sigma \Pi x{:}A.\ U : S} \qquad \frac{\Gamma \vdash_\Sigma U_1 : S \quad \Gamma \vdash_\Sigma U_2 : S \quad \vdash_\Sigma U_1 :: V \quad \vdash_\Sigma U_2 :: V}{\Gamma \vdash_\Sigma U_1 \,\&\, U_2 : S}$$

$$\frac{\Gamma \vdash_\Sigma U : \Pi x{:}A.\ V \qquad \Gamma \vdash_\Sigma M : A}{\Gamma \vdash_\Sigma U\, M : [M/x]V}$$

$$\frac{\vdash_\Sigma B :: A \qquad \Gamma \vdash_\Sigma A : \text{Type} \qquad \Gamma \vdash_\Sigma B : \text{Type} \qquad \Gamma, x{:}B \vdash_\Sigma M : C}{\Gamma \vdash_\Sigma \lambda x{:}A.\ M : \Pi x{:}B.\ C}(3)$$

$$\frac{\Gamma \vdash_\Sigma U : V \qquad V \equiv W \qquad \Gamma \vdash_\Sigma W : S}{\Gamma \vdash_\Sigma U : W}(4)$$

Note that we need a subsorting rule (2) *and* a type conversion rule (4), since we have formulated them as separate judgments which interact very little (formally). In the rule for λ-abstraction (3) one can see that the type label acts as a bound: we can analyze the expression for each sort B which refines A and conjoin the results using the introduction rule for & (1).

Finally, the rules for subsorting. The rules enforce the restriction that sorts and sort families can only be compared if they refine a common type.

$$\frac{\vdash_\Sigma U :: W \qquad \vdash_\Sigma V :: W}{\vdash_\Sigma U \,\&\, V \leq U} \qquad \frac{\vdash_\Sigma U :: W \qquad \vdash_\Sigma V :: W}{\vdash_\Sigma U \,\&\, V \leq V}$$

$$\frac{\vdash_\Sigma U \leq V_1 \qquad \vdash_\Sigma U \leq V_2}{\vdash_\Sigma U \leq V_1 \,\&\, V_2} \qquad \frac{a \leq b \text{ in } \Sigma}{\vdash_\Sigma a \leq b}$$

$$\frac{\vdash_\Sigma \Pi x{:}A.\ U_1 :: W \qquad \vdash_\Sigma \Pi x{:}A.\ U_2 :: W}{\vdash_\Sigma (\Pi x{:}A.\ U_1) \,\&(\Pi x{:}A.\ U_2) \leq (\Pi x{:}A.\ U_1 \,\&\, U_2)}$$

$$\frac{}{\vdash_\Sigma \mathrm{Type} \leq \mathrm{Type}} \qquad \frac{\vdash_\Sigma B \leq A \qquad \vdash_\Sigma U \leq V}{\vdash_\Sigma \Pi x{:}A.\ U \leq \Pi x{:}B.\ V} \qquad \frac{\vdash_\Sigma A \leq B}{\vdash_\Sigma A\, M \leq B\, M}$$

$$\frac{\vdash_\Sigma U :: W}{\vdash_\Sigma U \leq U} \qquad \frac{\vdash_\Sigma U \leq V \qquad \vdash_\Sigma V \leq W}{\vdash_\Sigma U \leq W}$$

The subsorting relationship is contravariant in the domain of a function type, as expected. Indexed sort families may only be compared if the indices are identical, which may require some applications of the type conversion rule (4) in a typing derivation before the subsumption rule (2) can be applied.

3.3 Properties of $\lambda^{\Pi\&}$

We begin by defining a forgetful mapping $\|.\|$ from $\lambda^{\Pi\&}$ to λ^{Π}. It ignores the distinctions introduced by sorts by collapsing them to the type they refine. The result of interpreting a signature Σ is a signature Σ' in λ^{Π} and a substitution σ mapping terms over Σ into terms over Σ'. We use $\sigma(U)$ as a notation for the result of applying σ to U with the special provision that

$$\sigma(U_1 \,\&\, U_2) = \begin{cases} V & \text{if } \sigma(U_1) = \sigma(U_2) = V \\ \text{undefined} & \text{otherwise} \end{cases}$$

The application of σ to a context Γ distributes into the constituent terms. The empty substitution is denoted by $[]$ and the extension of a substitution σ mapping the new constant d to d' is written as $\sigma \oplus [d \mapsto d']$.

$$\begin{array}{rcll} \|\cdot\| & = & \langle \cdot; [] \rangle & \\ \|\Sigma, d{:}U\| & = & \|\Sigma\| & \text{if } d \text{ declared in } \Sigma \\ \|\Sigma, d{:}U\| & = & \langle \Sigma', d{:}\sigma(U); \sigma \oplus [d \mapsto d] \rangle & \text{if } d \text{ not declared in } \Sigma \text{ and } \langle \Sigma'; \sigma \rangle = \|\Sigma\| \\ \|\Sigma, a_1 :: a_2\| & = & \langle \Sigma'; \sigma \oplus [a_1 \mapsto a_2] \rangle & \text{where } \langle \Sigma'; \sigma \rangle = \|\Sigma\| \\ \|\Sigma, a_1 \leq a_2\| & = & \|\Sigma\| & \end{array}$$

Lemma 1 *If Σ is valid and $\vdash_\Sigma U \leq V$ then there exists a (unique) W such that $\vdash_\Sigma U :: W$ and $\vdash_\Sigma V :: W$.*

Lemma 2 (Refinement) *Let Σ be a valid signature, Γ be a valid context, and $\|\Sigma\| = \langle \Sigma'; \sigma \rangle$. Then:*

(i) *if $\vdash_\Sigma U :: V$ then $\sigma(U) = \sigma(V)$,* *(ii)* *if $\vdash_\Sigma U \leq V$ then $\sigma(U) = \sigma(V)$,*
(iii) *if $U \equiv V$ then $\sigma(U) \equiv \sigma(V)$,* *(iv)* *if $\Gamma \vdash_\Sigma U : V$ then $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(U) : \sigma(V)$.*

Proof: By straightforward inductions over the derivations of the given judgments, employing uniqueness of bounds and Lemma 1. $\square$

We call a $\lambda^{\Pi\&}$ term *canonical* if it is in long $\beta\eta$-normal form, as in LF.

Lemma 3 *The judgment $U \equiv V$ is decidable on valid terms and every valid term U has a unique equivalent canonical form.*

Proof sketch: The corresponding judgment on LF is decidable on valid LF terms (see, for example, [Geu92]). Equivalence on types and kinds is structural and therefore trivially decidable, except for conversions among the embedded objects. But labels of λ-abstractions are restricted to terms which remain unchanged under the forgetful interpretation, and thus conversions in $\sigma(U)$ and $\sigma(V)$ can be lifted to conversions in U and V. $\square$

The equivalence relation $\cong$ is defined by $U \cong V$ iff $U \leq V$ and $V \leq U$. It is easily shown that this is a congruence. Also, the following properties are easily proved.

Lemma 4 (Basic Properties of Sorts) *We assume implicitly that both sides of each of the equivalences below refine the same type.*

$$\begin{array}{llll} (i) & U \,\&\, V \cong V \,\&\, U, & (ii) & U \,\&(V \,\&\, W) \cong (U \,\&\, V) \,\&\, W, \\ (iii) & U \,\&\, U \cong U, & (iv) & (\Pi x{:}A.\ U_1) \,\&(\Pi x{:}A.\ U_2) \cong (\Pi x{:}A.\ U_1 \,\&\, U_2). \end{array}$$

Theorem 5 (Decidability of Subsorting) *The subsorting judgment $\vdash_\Sigma U \leq V$ is decidable for valid signatures Σ.*

Proof sketch: By an interpretation into the subtyping problem for Forsythe, for which a decidability proof has been given by Reynolds [personal communication, 1991]. The proof can be found in [Pie91] in a slightly different form. Each atomic type of the form $a\, M_1 \ldots M_n$ is interpreted as a simple type $\overline{a\, M_1 \ldots M_n}$ which inherits its subsorting property from a. The main observation in the correctness proof of this interpretation is that $A\, M \leq B\, N$ iff $A \leq B$ and $M = N$. $\square$

We call a type A a *minimal type* for the object M in context Γ if A is canonical and for every canonical B such that $\Gamma \vdash_\Sigma M : B$ we have $\vdash_\Sigma A \leq B$. A similar definition applies to minimal kinds.

Theorem 6 (Decidability of $\lambda^{\Pi\&}$) *The validity of signatures and contexts and the typing judgment $\Gamma \vdash_\Sigma U : V$ are decidable. Furthermore, every valid term U has a minimal type or kind.*

Proof sketch: Using the forgetful interpretation and the soundness and completeness of the algorithmic version of LF in [HHP93] we can show that each derivation can be transformed into one which eagerly applies normalization on types, but otherwise requires no type conversion. Secondly we show that applications of the subsorting rule in such a derivation can be pushed up to the leaves, except for λ-abstractions and applications, where we can directly calculate a minimal type from minimal types of the constituents. The completeness of this calculation relies on the fact that only finitely many sorts (modulo $\cong$) refine a given type. $\square$

4 Examples Revisited

Now that the $\lambda^{\Pi\&}$ calculus has been defined, we revisit the earlier examples. We use the concrete syntax `::` for :: and `<:` for $\leq$.

Hereditary Harrop formulas. Following the previous and unchanged definitions of the connectives, we declare atoms, goals, and programs as refinements of formulas. Then we declare sorts for the constructors.

```
atom :: form.  % atoms
goal :: form.  % legal goals
prog :: form.  % legal programs

atom <: goal.  % every atom is a legal goal

=>   : prog -> goal -> goal.
||   : goal -> goal -> goal.
&&   : goal -> goal -> goal.

atom <: prog.  % every atom is a legal program

=>   : goal -> prog -> prog.
&&   : prog -> prog -> prog.
```

The entailment and backchaining judgments can now be declared naturally. Their definition (not shown here) is also simple and intuitive.

```
solve     : prog -> goal -> type.
backchain : prog -> atom -> goal -> type.
```

Normal Natural Deductions. Here, both *elim* and *nf* become sort families which refine *pf*. Following the previous declarations for *pf*, *impi*, and *impe* we complete the definition as follows.

```
nf   :: pf.  % normal form deductions
elim :: pf.  % pure elimination deductions from hypotheses

elim <: nf.  % every elim deduction is in normal form

impi : (elim A -> nf B) -> nf (imp A B).
impe : elim (imp A B) -> nf A -> elim B.
```

Below we show the obvious deduction of $p \supset (q \supset p)$ for parameters p and q. Terms of the form $\lambda x{:}A.\ M$ are written as `[x:A] M` in concrete syntax.

```
   ([p:o] [q:o] impi ([P:pf p] impi ([Q:pf q] P))
 : {p:o} {q:o} nf (imp p (imp q p)).
```

These small examples should help to illustrate how refinement types provide a natural and direct means to express subtyping in the context of a logical framework. Many of the case studies of deductive systems in LF that we and others have carried out would benefit similarly.

5 Conclusion and Further Work

We plan to implement the system $\lambda^{\Pi\&}$ as an extension of Elf. This requires a generalization of the constraint solving algorithm in [KP93] to dependent types, and the development of a feasible type reconstruction algorithm. The type-checking algorithm which arises out of the proof of Theorem 6 works by bottom-up synthesis and is not practical. However, a top-down type-checking algorithm as in the implementation of refinement types for ML [FP91] promises to be of acceptable efficiency, especially since our language lacks recursion at the level of terms.

We would also like to consider relaxing some of the restrictions currently in place to enforce orthogonality of conversion and subsorting. In particular, it is intuitively appealing to allow sorts (to be interpreted as bounds) in the labels of λ-abstractions, but we believe that this necessitates a form of typed or sorted conversion and our decidability proof no longer applies directly. This slightly different version of $\lambda^{\Pi\&}$ also appears to be better suited for an extension to the Calculus of Constructions with refinement types. It is consistent with our system to allow *refinement kinds*, that is, declarations of the form $k :: \text{Type}$. This leads to a system which encompasses ELF^{+} [Gar92] and could also yield a new view of type classes in the context of type theory. We plan to investigate the meta-theoretic properties of a type theory with refinement types and refinement kinds.

One might also consider promotion of sorts to types and demotion of types to sorts which sometimes further economizes representations without making them less intuitive. We plan to investigate this in the context of the module system for LF described in [HP99].

Finally, there is the question of adequacy proofs for representations in $\lambda^{\Pi\&}$. The normal form theorem is useful here, but we would also like to give an interpretation which maps a signature in $\lambda^{\Pi\&}$ into an equivalent signature in λ^{Π}. We conjecture that there is such a mapping which interprets refinement by relativizing Π-quantifiers and subsorting by coercions.

Acknowledgments. We would like to thank Michael Kohlhase and Tim Freeman for discussions regarding refinement types and Nevin Heintze, Benjamin Pierce, and Ekkehard Rohwedder for comments on a draft of this extended abstract.

Appendix: The λ-Cube

In this appendix we give a uniform and very elegant presentation of Barendregt's λ-cube and in particular of LF and the calculus of constructions in which the levels (objects, families, kinds) are *refinements* of a proper type of terms. This example also shows why it is useful to allow $K_1 \mathbin{\&} K_2$ in $\lambda^{\Pi\&}$. We omit the rules for type conversion for the sake of brevity.

```
term : type.

tp : term.
pi : term -> (term -> term) -> term.
lm : term -> (term -> term) -> term.
ap : term -> term -> term.

%% Levels

sup :: term.  % super-kind
knd :: term.  % kinds
fam :: term.  % families
```

```
obj :: term.  % object

%% The LF declarations.

tp : sup.

tp : knd.
pi : fam -> (obj -> knd) -> knd.

pi : fam -> (obj -> fam) -> fam.
lm : fam -> (obj -> fam) -> fam.
ap : fam -> obj -> fam.

lm : fam -> (obj -> obj) -> obj.
ap : obj -> obj -> obj.
```

In order to obtain the calculus of construction, we add the following declarations.

```
pi : knd -> (fam -> knd) -> knd.

lm : knd -> (fam -> fam) -> fam.
ap : fam -> fam -> fam.
pi : knd -> (fam -> fam) -> fam.

lm : knd -> (fam -> obj) -> obj.
ap : obj -> fam -> obj.
```

The typing judgment is now uniform across the levels.

```
of : knd -> sup -> type
   & fam -> knd -> type
   & obj -> fam -> type.

of_tp : of tp tp.

of_pi : of (pi T1 T2) tp
      <- of T1 tp
      <- {x:term} of x T1 -> of (T2 x) tp.

of_lm : of (lm T1 T2) (pi T1 T3)
      <- of T1 tp
      <- {x:term} of x T1 -> of (T2 x) (T3 x).

of_ap : of (ap T1 T2) (T4 T2)
      <- of T1 (pi T3 T4)
      <- of T2 T3.
```

References

[CDCV81] Mario Coppo, Maria Dezani-Ciancaglini, and B. Venneri. Functional character of solvable terms. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 27:45–58, 1981.

[CG92] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In J.-C. Raoult, editor, *17th Colloquium on Trees in Algebra and Programming, Rennes, France*, pages 102–123, Berlin, February 1992. Springer-Verlag LNCS 581.

[Fel89] Amy Felty. *Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, July 1989. Available as Technical Report MS-CIS-89-53.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[Gar92] Philippa Gardner. *Representing Logics in Type Theory.* PhD thesis, University of Edinburgh, July 1992. Available as Technical Report CST-93-92.

[Geu92] Herman Geuvers. The Church-Rosser property for $\beta\eta$-reduction in typed λ-calculi. In A. Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 453–460, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Hay91] Susumu Hayashi. Singleton, union and intersection types for program extraction. In T. Ito and A. R. Meyer, editors, *Proceedings of the International Conference on Theoretical Aspects of Software*, pages 701–730, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HP99] Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Functional Programming*, 199? To appear.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Koh92] Michael Kohlhase. Unification in order-sorted type theory. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, pages 421–432, St. Petersburg, Russia, July 1992. Springer-Verlag LNAI 624.

[KP93] Michael Kohlhase and Frank Pfenning. Unification in a λ-calculus with intersection types. In Dale Miller, editor, *Proceedings of the International Logic Programming Symposium*, Vancouver, Canada, October 1993. MIT Press. To appear.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[NQ92] Tobias Nipkow and Zhenyu Qian. Reduction and unification in lambda calculi with subtypes. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, pages 66–78, Saratoga Springs, New York, June 1992. Springer-Verlag LNAI 607.

[Pfe91a] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe91b] Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 74–85, Amsterdam, The Netherlands, July 1991.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1991. Available as Technical Report CMU–CS–91–205.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical Report 189, Computer Laboratory, University of Cambridge, January 1990.

[Rey88] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1988.

[Rey91] John C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, pages 675–700, Sendai, Japan, September 1991. Springer-Verlag LNCS 526.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* Dissertation, Universität Kaiserslautern, May 1989.

[SS88] Anne Salvesen and Jan M. Smith. The strength of the subset type in Martin-Löf's type theory. In *Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland*, pages 384–391. IEEE, July 1988.

[SS89] Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations.* Springer-Verlag LNAI 395, 1989.

Closure Under Alpha-Conversion

Randy Pollack
LFCS, University of Edinburgh

Sept. 1993

1 Introduction

Consider an informal presentation of simply typed λ-calculus as in [Bar92]. Leaving out some of the details, let σ, τ range over simple types, x, y range over a class of term variables, and M, N, range over the Church-style terms. A *statement* has the form $M : \sigma$, where M is the *subject* of the statement and σ is its *predicate*. A *context*, ranged over by Γ, is a list of statements with only variables as subjects. A context is *valid* if it contains only distinct variables as subjects. A statement, $M : \sigma$, is *derivable* from valid context Γ, notation $\Gamma \vdash M : \sigma$, if $\Gamma \vdash M : \sigma$ can be produced using the following rules.

$$\text{VAR} \qquad \Gamma \vdash x : \sigma \qquad \text{if } (x{:}\sigma) \in \Gamma$$

$$\text{LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash [x{:}\sigma]M : \sigma \to \tau}$$

$$\text{APP} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

Such a presentation is usually considered formal enough for everyday reasoning. It can be implemented literally as a type synthesis algorithm for $\lambda\!\to$: to compute a type for a variable, look it up in the context; to compute a type for a lambda abstraction, compute a type for its body in an extended context; to compute a type for an application, compute types for its left and right components, and check that they match appropriately. Now lets use the algorithm to compute a type for $a = [x{:}\tau][x{:}\sigma]x$.

$$\cfrac{\cfrac{x{:}\tau, x{:}\sigma \vdash x : \tau}{x{:}\tau \vdash [x{:}\sigma]x : \sigma \to \tau}}{\vdash [x{:}\tau][x{:}\sigma]x : \tau \to \sigma \to \tau}$$

Surely this is not right; the type should be $\tau \to \sigma \to \sigma$ because we intended the rightmost x to be bound by the second lambda, not by the first lambda. What went wrong? In directly implementing this system we are taking informal notation more literally than intended: there is no "x" in $[x{:}\tau][x{:}\sigma]x$; the names of bound variables are not meant to be taken seriously. The rule LDA should be read as "in order to type $[x{:}\sigma]M$, choose some suitable alpha-representative of $[x{:}\sigma]M$, ...".

Here is a more concrete presentations of $\lambda\!\to$. First we state what it means to be a valid context:

$$\text{NIL-VALID} \qquad \bullet\ \mathsf{valid}$$

$$\text{CONS-VALID} \qquad \frac{\Gamma\ \mathsf{valid}}{\Gamma, x{:}\sigma\ \mathsf{valid}} \qquad x \notin \mathsf{Dom}\,(\Gamma)$$

Now we can fix the informal system in several ways. The most conservative approach is to change the VAR rule to look up a variable in a valid context:

$$\text{VAR} \qquad \frac{\Gamma\ \mathsf{valid}}{\Gamma \vdash x : \sigma} \qquad x{:}\sigma \in \Gamma$$

$$\text{LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash [x{:}\sigma]M : \sigma \to \tau}$$

$$\text{APP} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\,N : \tau}$$

This system does not derive any type for a:

$$\cfrac{\cfrac{\cfrac{\cfrac{\text{FAILURE: no rule applies because } x \in \mathsf{Dom}\,(x{:}\tau)}{x{:}\tau, x{:}\sigma\ \mathsf{valid}}}{x{:}\tau, x{:}\sigma \vdash x : ?}}{x{:}\tau \vdash [x{:}\sigma]x : \sigma \to ?}}{\vdash [x{:}\tau][x{:}\sigma]x : \tau \to \sigma \to ?} \tag{1}$$

While it doesn't derive the unintended type $\tau \to \sigma \to \tau$, this system also fails to derive the intended type $\tau \to \sigma \to \sigma$. Notice that $\vdash [x{:}\tau][y{:}\sigma]y : \tau \to \sigma \to \sigma$ is derivable, but the system is not closed under alpha-conversion of subjects.

Formal systems with variable binding are implemented on machines as the basis of programming languages and proof checkers, among other applications. It is clear that the concrete syntax that users enter into such implementations, and see printed by the implementation in response, should be formally related to the implemented formal system. Further, users and implementors need an exact and concise description of such a system; informal explanation is not good enough.

The concrete syntax should have good properties. Users of such implemented systems will construct large formal objects with complex binding. The implementation should help in this task, and anamolies of naming such as the small example above make the job more difficult. What do you say to an ML implementation that claims `fn x => fn x => x` is not well typed?

There are several approaches to naming in implementations of formal systems. Perhaps the best known is the use of explicit names, and Curry-style renaming in the definition of substitution. This technique can (probably) be formalized. The difficulty arises when we ignore the distinction between alpha-convertible terms, and treat then as equal.

It is well known that one solution to the problems of alpha-conversion is the use of de Bruijn "nameless variables" [dB72]. Although nameless variables have their partisans for use in metatheoretic study, even those partisans admit that the explanation of substitution of a term for a given variable is painful in such a presentation, although it can be, and has been, carried out elegantly [Alt93, Hue93][1]. However, the direct use of nameless variables is not a real possibility in pragmatic applications because human users find it difficult to write even small expressions using nameless variables. It is necessary to translate from named syntax to nameless, and then back again to named syntax for pretty printing, and this translation itself must be formalized.

I know of two recent proposals that take names seriously, but avoid the need for alpha-conversion. One proposal, by Coquand [Coq91], follows a style in logic to distinguish between free variables (parameters) and bound variables (variables). This idea has been used to formalize a large theory of Pure Type Systems, including reduction, conversion and typing [MP93]. This formalization does distinguish between alpha-convertible terms, and the typing judgement is indeed closed under alpha-conversion. The other proposal, by Martin-Löf [Tas93] goes much further, not only using explicit names, but also explicit substitutions, i.e. making the notion of substitution a part of the formal system (as originally proposed for nameless terms in [ACCL91]). Unfortunately the system of [Tas93], in its current formulation, is not closed under alpha-conversion.

Finally, there is a recent proposal [Gor93] of a formalization mixing nameless terms and named variables in such a way that named terms are equal up to alpha-conversion.

1.1 The Constructive Engine

The Constructive Engine [Hue89] is an abstract machine for type checking the Calculus of Constructions. It is the basis for the proofcheckers Coq [DFH+93] and LEGO [LP92]. Among its interesting aspects are:

1. converting the non-deterministic typing rules of the underlying type theory into a deterministic, syntax-directed program
2. optimizations in the size of derivations
3. translating from external *concrete* syntax, with explicit variable names, into internal *abstract* syntax of *locally nameless* terms, that is, local binding by de Bruijn indexes, and global binding by explicit names
4. an efficient technique for testing conversion of locally nameless terms (with special attention to the treatment of definitions)

The basic ideas of the first and second of these points appeared in early writing of Martin-Löf. The first point has been studied extensively [Pol92, vBJMP93] for the class of Pure Type Systems. The fourth point (which is clearly the limiting factor in pragmatic implementations of proof checkers) has not received any theoretical attention to my knowledge, although [Hue89, dB85] have interesting ideas.

In this note I will discuss the third item above, the relationship between concrete syntax and abstract syntax in the Constructive Engine. The use of locally nameless style for internal representation of terms is one of the basic decisions of the Constructive Engine, but perhaps reflects more Huet's interest in experimenting with de Bruijn representation than any ultimate conviction that they are the "right" notation for implementing a type checker. I do not want to study the pros and cons of this representation for efficient typechecking, but only to muse over

[1] For an example of metatheory where nameless variables are very inconvenient, see the discussion of the Thinning Lemma in [MP93]

the relationship between concrete terms, their abstract representations, and their (abstract) types.

In the next section we discuss simply-typed lambda calculus, $\lambda\!\to$. After presenting several concrete and abstract presentations of its typing rules, we derive a Constructive Engine for $\lambda\!\to$, and consider some variations.

In section 3 we consider the same issue for Pure Type Systems (PTS). There is one new problem in the case of dependent types. We explain a constructive engine for dependent types, and show how to make it closed under alpha-conversion of terms.

2 Simply typed lambda calculus

There is a crude way to close the relation $\vdash$ of the Introduction under alpha-conversion of subjects, by adding a rule

$$\text{ALPHA} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash N : \sigma} \qquad M \overset{\alpha}{=} N$$

where $M \overset{\alpha}{=} N$, alpha-conversion, must also be defined by some inductive definition. Such solutions are heavy, and not ideal for either implementation or formal meta-reasoning. Instead of reasoning about three or five rules, we'll have to reason about all the rules for $\overset{\alpha}{=}$ as well. Further, rules such as ALPHA, that are not syntax-directed, are hard to reason about: being non-deterministic, they allow many derivations of the same judgement, which sometimes prevents proof by induction on the structure of derivations.

2.1 A concrete presentation closed under alpha-conversion

Another approach is to formalize the informal meaning of the LDA rule suggested above: choose a sufficiently fresh variable name to substitute for x. Informally, replace LDA by

$$\frac{\Gamma, y{:}\sigma \vdash_{\mathsf{S}} M[y/x] : \tau}{\Gamma \vdash_{\mathsf{S}} [x{:}\sigma]M : \sigma \to \tau} \qquad y \notin M$$

Substitution of y for x in M must still be defined, and the usual definition involves alpha-conversion. We give a formulation suggested in [Coq91] and formalized in detail in [MP93]. Let p, q, r, range over an infinite set of *parameters*, and x, y, z over variables as before. Parameters and variables are disjoint sets[2]. Define two operations of replacement. Replacing a parameter by a term is entirely textual:

$$\begin{array}{rcl} x[M/p] & = & x \\ q[M/p] & = & \text{if } p = q \text{ then } M \text{ else } q \\ ([x{:}\sigma]N)[M/p] & = & [x{:}\sigma]N[M/p] \\ (N_1\, N_2)[M/p] & = & (N_1[M/p])\,(N_2[M/p]) \end{array}$$

[2] Another informality! What is required is that the *terms* p and x be distinguishable, not necessarily that the underlying objects be. Depending on the formalization of terms we might use weaker conditions

Replacing a variable by a term does respect the scope of variable binding but does not rename variables to prevent capture:

$$\begin{array}{rcl} x[M/y] & = & \text{if } y = x \text{ then } M \text{ else } x \\ q[M/y] & = & q \\ ([x{:}\sigma]N)[M/y] & = & [x{:}\sigma](\text{if } y = x \text{ then } N \text{ else } N[M/y]) \\ (N_1\, N_2)[M/y] & = & (N_1[M/y])\,(N_2[M/p]) \end{array}$$

Now replace LDA by

$$\text{S-LDA} \qquad \frac{\Gamma, p{:}\sigma \vdash_{\mathsf{s}} M[p/x] : \tau}{\Gamma \vdash_{\mathsf{s}} [x{:}\sigma]M : \sigma \to \tau} \qquad p \notin M$$

(where $p \in M$ means *textual* occurrence). In the side condition of this rule, it's not necessary to check that $p \notin \mathsf{Dom}\,(\Gamma)$ because failure of that condition prevents completing a derivation; just choose another parameter, since p does not occur in the conclusion of the rule. It *is* necessary to check $p \notin M$ so that the premiss doesn't bind instances of p that do not arise from x.

We could define beta-reduction, beta-conversion, prove Church-Rosser, subject reduction, etc. [MP93], but for our purposes alpha-conversion is enough:

$$\alpha\text{-REFL} \qquad M \stackrel{\alpha}{=} M$$

$$\alpha\text{-LDA} \qquad \frac{M[p/x] \stackrel{\alpha}{=} M'[p/y]}{[x{:}\sigma]M \stackrel{\alpha}{=} [y{:}\sigma]M'} \qquad p \notin M,\ \notin M'$$

$$\alpha\text{-APP} \qquad \frac{M \stackrel{\alpha}{=} M' \qquad N \stackrel{\alpha}{=} N'}{M\,N \stackrel{\alpha}{=} M'\,N'}$$

Now we can state and prove $\vdash_{\mathsf{s}}$ is closed under alpha-conversion

Lemma 1 (Closure of $\vdash_{\mathsf{s}}$ under alpha-conversion)
If $\Gamma \vdash_{\mathsf{s}} M : \sigma$ and $M \stackrel{\alpha}{=} M'$ then $\Gamma \vdash_{\mathsf{s}} M' : \sigma$

The proof follows the same outline as a proof of subject reduction (closure under beta-reduction).

$\vdash_{\mathsf{s}}$ still treats parameters seriously: $\vdash_{\mathsf{s}} [x{:}\tau][x{:}\sigma]x : \tau \to \sigma \to \sigma$ but $P{:}\tau, P{:}\sigma \not\vdash_{\mathsf{s}} P : \sigma$. This is a different problem, if it's a problem at all. In $\vdash_{\mathsf{s}}$ we have analysed the transition from local variable to global parameter, while the treatment of parameters themselves is the same as in $\vdash$.

An interesting variation on the previous idea is to use generalized induction to truly remove the fresh name from derivations. Replace LDA by

$$\text{G-LDA} \qquad \frac{\forall p \notin \mathsf{Dom}\,(\Gamma)\ .\ \Gamma, p{:}\sigma \vdash_{\mathsf{g}} M[p/x] : \tau}{\Gamma \vdash_{\mathsf{g}} [x{:}\sigma]M : \sigma \to \tau}$$

Notice again the "side condition", this time appearing as an antecedent of the generalized premiss. We don't exclude those p that happen to occur in M, because we must derive $\Gamma, p{:}\sigma \vdash_{\mathsf{g}} M[p/x] : \tau$ for infinitely many p (using that the class of variables is infinite), while M can contain only finitely many p. On the other hand, for $p \in \mathsf{Dom}\,(\Gamma)$, $\Gamma, p{:}\sigma \vdash_{\mathsf{g}} M[p/x] : \tau$ is not derivable for reasons having nothing to do with M or τ.

Both $\vdash_{\mathsf{s}}$ and $\vdash_{\mathsf{g}}$ derive more judgements than $\vdash$; in fact $\vdash_{\mathsf{s}}$ and $\vdash_{\mathsf{g}}$ are equivalent. Since this is not completely obvious (and because I have a better proof than previuosly published in [MP93]) I will prove it here.

Lemma 2 *For all* Γ*,* M *and* σ*,*
$\Gamma \vdash_{\mathsf{s}} M : \sigma \quad \Leftrightarrow \quad \Gamma \vdash_{\mathsf{g}} M : \sigma$

Proof The direction $\Leftarrow$ is trivial by induction on a derivation of $\Gamma \vdash_{\mathsf{g}} M : \sigma$.

In order to prove direction $\Rightarrow$ we introduce the machinary of renaming. A *renaming* (ranged over by ϕ) is an almost-everywhere-identity function from parameters to parameters. We extend the action of renamings compositionally to terms and contexts. It's easy to see that *bijective* renamings respect both $\vdash_{\mathsf{s}}$ and $\vdash_{\mathsf{g}}$; in particular, if ϕ is bijective and $\Gamma \vdash_{\mathsf{g}} M : \sigma$, then $\phi\Gamma \vdash_{\mathsf{g}} \phi M : \sigma$. It's a little difficult to construct bijective renamings in general, because they must be almost-everywhere-identity, i.e. not only the parameters that get moved have to be considered, but also those that are fixed; a combinatorial nightmare. However it's clear that any renaming that only swaps parameters, e.g. $\{q \mapsto p,\ p \mapsto q\}$, is bijective.

Now we prove $\Gamma \vdash_{\mathsf{g}} M : \sigma$ by structural induction on a derivation of $\Gamma \vdash_{\mathsf{s}} M : \sigma$. All cases are trivial except the rule S-LDA

$$\textsc{s-lda} \qquad \frac{\Gamma, p{:}\sigma \vdash_{\mathsf{s}} M[p/x] : \tau}{\Gamma \vdash_{\mathsf{s}} [x{:}\sigma]M : \sigma \to \tau} \qquad p \notin M$$

By induction hypothesis $\Gamma, p{:}\sigma \vdash_{\mathsf{g}} M[p/x] : \tau$. In order to show $\Gamma \vdash_{\mathsf{g}} [x{:}\sigma]M : \sigma \to \tau$ by G-LDA we only need to show $\forall r \notin \mathsf{Dom}\,(\Gamma)\ .\ \Gamma, r{:}\sigma \vdash_{\mathsf{g}} M[r/x] : \tau$, so

$$\text{let} \quad r \notin \mathsf{Dom}\,(\Gamma) \quad \text{and show} \quad \Gamma, r{:}\sigma \vdash_{\mathsf{g}} M[r/x] : \tau.$$

Taking $\phi = \{p \mapsto r,\ r \mapsto p\}$, we have $\phi(\Gamma, p{:}\sigma) \vdash_{\mathsf{g}} \phi(M[p/x]) : \tau$ is derivable by renaming the induction hypothesis. Thus we are finished if we can show

$$\phi(\Gamma, p{:}\sigma) = \Gamma, r{:}\sigma \qquad \text{and} \qquad \phi(M[p/x]) = M[r/x].$$

Notice $p \notin \mathsf{Dom}\,(\Gamma)$ (or the premiss of S-LDA could not be derivable), and we also know $p \notin M$ and $r \notin \mathsf{Dom}\,(\Gamma)$. From these observations it's clear that the first equation holds. For the second equation, notice that if $r = p$ then ϕ is the identity renaming, and we are done, so assume $r \neq p$, and hence $r \notin M[p/x]$ (again, from the premiss of S-LDA). Now

$$\phi(M[p/x]) = \{p \mapsto r\}(M[p/x]) = (\{p \mapsto r\}M)[\{p \mapsto r\}p/x]) = M[r/x]$$

as required. ∎

You may wonder why we are interested in $\vdash_{\mathsf{g}}$ given that it has the same judgements as $\vdash_{\mathsf{s}}$, but is clearly less "concrete", involving infinitely branching trees as it does. In fact $\vdash_{\mathsf{g}}$ allows some arguments by induction over the structure of derivations that I do not know how to do using $\vdash_{\mathsf{s}}$. The proof of the thinning lemma for Pure Type Systems detailed in [MP93] is an example of this, and there are many others in the work outlined in that paper. Its clear that while there are infinitely many derivations of, for example, $\vdash_{\mathsf{s}} [x{:}\sigma]x : \sigma \to \sigma$, each containing some particular variable not occurring in the conclusion, $\vdash_{\mathsf{g}} [x{:}\sigma]x : \sigma \to \sigma$ has only one derivation that does not not depend on any particular variable not occurring in the conclusion. Loosly speaking, $\vdash_{\mathsf{g}}$ has a "subformula property" that $\vdash_{\mathsf{s}}$ lacks.

2.2 Some other systems for $\lambda\!\to$

We are now working towards a Constructive Engine for $\lambda\!\to$, and this will be a system for typing concrete terms with named variables that is closed under alpha-conversion. First we present an optimization that suggests a new idea for handling global variables. Following this idea we will see a presentation of $\lambda\!\to$ not mentioning substitution, that is closed under alpha-conversion.

An optimization I think this optimization first appears in early writing of Martin-Löf, and it is also used in the Constructive Engine. It is interesting for our present purposes because it distinguishes between the variables that have always been global, i.e. bound by the context part of the judgement, and those that have only "locally" become global during construction of a derivation.

The inductive definition of $\vdash$ is very inefficient in duplicating the test that Γ valid on each branch of a derivation. For example

$$\dfrac{\dfrac{\vdots \atop x{:}\sigma\to\sigma, y{:}\sigma\ \mathsf{valid}}{x{:}\sigma\to\sigma, y{:}\sigma \vdash x : \sigma\to\sigma} \qquad \dfrac{\vdots \atop x{:}\sigma\to\sigma, y{:}\sigma\ \mathsf{valid}}{x{:}\sigma\to\sigma, y{:}\sigma \vdash y : \sigma}}{x{:}\sigma\to\sigma, y{:}\sigma \vdash x\ y : \sigma}$$

We can optimize $\vdash$ by moving the test for a valid context outside the typing derivation, that is, test once and for all that the given context is in fact valid, and then whenever a derivation extends the context (using the LDA rule), check that the extension preserves validity.

O-VAR	$\Gamma \vdash_{\mathsf{o}} x : \sigma$	$x{:}\sigma \in \Gamma$
O-LDA	$\dfrac{\Gamma, x{:}\sigma \vdash_{\mathsf{o}} M : \tau}{\Gamma \vdash_{\mathsf{o}} [x{:}\sigma]M : \sigma\to\tau}$	$x \notin \mathsf{Dom}\,(\Gamma)$
O-APP	$\dfrac{\Gamma \vdash_{\mathsf{o}} M : \sigma\to\tau \qquad \Gamma \vdash_{\mathsf{o}} N : \sigma}{\Gamma \vdash_{\mathsf{o}} M\,N : \tau}$	

Comparing with $\vdash$, notice that O-VAR does not check that Γ is valid, but O-LDA does maintain this property during derivations.

The sense in which $\vdash_{\mathsf{o}}$ is correct is given by

Lemma 3 (Correctness of $\vdash_{\mathsf{o}}$ for $\vdash$)
$\Gamma \vdash M : \sigma \quad \Leftrightarrow \quad (\Gamma\ \mathsf{valid}\ \textit{and}\ \Gamma \vdash_{\mathsf{o}} M : \sigma)$

Proof The direction $\Rightarrow$ is trivial by induction on a derivation of $\Gamma \vdash M : \sigma$.

The direction $\Leftarrow$ says there is a terminating algorithm for putting back all the redundant information removed from a $\vdash$-derivation[3]. For systems of dependent types, where correctness of a context and the typing judgement are mutually inductive, this is not completely trivial. The present case can be proved by induction on the structure of a derivation of $\Gamma \vdash_{\mathsf{o}} M : \sigma$. ∎

In the tiny example above, we only need to check

$$\vdots \atop x{:}\sigma\to\sigma, y{:}\sigma\ \mathsf{valid} \qquad \text{and} \qquad \dfrac{x{:}\sigma\to\sigma, y{:}\sigma \vdash_{\mathsf{o}} x : \sigma\to\sigma \qquad x{:}\sigma\to\sigma, y{:}\sigma \vdash_{\mathsf{o}} y : \sigma}{x{:}\sigma\to\sigma, y{:}\sigma \vdash_{\mathsf{o}} x\ y : \sigma}$$

[3] I owe this observation to Stefano Berardi.

The root of the problem We can view the systems presented so far as leaving unspecified how a context is searched for the type of a variable. This leaves open the possibility for various implementations, such as linear search, or hash-coding. The price we pay is the requirement for only one binding occurrence of a variable in a valid context. This is actually the root of our problem about closure under alpha-conversion, since we don't restrict to only one binding instance for a variable in a term. Informally our idea is to replace the rule O-VAR, which does not specify how to search Γ for an assignment to x, by

$$\Gamma \vdash x : \mathsf{assoc}\; x\; \Gamma$$

which searches Γ linearly ($\mathsf{assoc}\; x\; \Gamma$ returns the type of the *first* occurrence of x in Γ, viewing Γ as a list that conses on the right). More precisely, we replace O-VAR with two rules that search Γ from right to left.

I-START	$\Gamma, x{:}\sigma \vdash_{\mathsf{i}} x : \sigma$	
I-WEAK	$\dfrac{\Gamma \vdash_{\mathsf{i}} x : \sigma}{\Gamma, y{:}\tau \vdash_{\mathsf{i}} x : \sigma}$	$x \neq y$
I-LDA	$\dfrac{\Gamma, x{:}\sigma \vdash_{\mathsf{i}} M : \tau}{\Gamma \vdash_{\mathsf{i}} [x{:}\sigma]M : \sigma \to \tau}$	$x \notin \mathsf{Dom}\,(\Gamma)$
I-APP	$\dfrac{\Gamma \vdash_{\mathsf{i}} M : \sigma \to \tau \qquad \Gamma \vdash_{\mathsf{i}} N : \sigma}{\Gamma \vdash_{\mathsf{i}} M\, N : \tau}$	

$\vdash_{\mathsf{i}}$ has fewer judgements than $\vdash_{\mathsf{o}}$, for example $x{:}\sigma, x{:}\tau \vdash_{\mathsf{o}} x : \sigma$ but $x{:}\sigma, x{:}\tau \not\vdash_{\mathsf{i}} x : \sigma$. However only judgements of $\vdash_{\mathsf{o}}$ that are incorrect for $\vdash$ are excluded: if Γ is valid, the order of search doesn't matter.

Lemma 4 (Correctness of $\vdash_{\mathsf{i}}$ for $\vdash$) *If* Γ valid *then*
$\Gamma \vdash_{\mathsf{o}} M : \sigma \quad \Leftrightarrow \quad \Gamma \vdash_{\mathsf{i}} M : \sigma$

Closure under alpha-conversion The system $\vdash_{\mathsf{i}}$ seems strange: why would we be more operational in presenting an abstract relation than required? The payoff is that now the side condition $x \notin \mathsf{Dom}\,(\Gamma)$ of rule I-LDA can also be dropped, giving the system of "liberal terms":

LT-START	$\Gamma, x{:}\sigma \vdash_{\mathsf{lt}} x : \sigma$	
LT-WEAK	$\dfrac{\Gamma \vdash_{\mathsf{lt}} x : \sigma}{\Gamma, y{:}\tau \vdash_{\mathsf{lt}} x : \sigma}$	$x \neq y$
LT-LDA	$\dfrac{\Gamma, x{:}\sigma \vdash_{\mathsf{lt}} M : \tau}{\Gamma \vdash_{\mathsf{lt}} [x{:}\sigma]M : \sigma \to \tau}$	
LT-APP	$\dfrac{\Gamma \vdash_{\mathsf{lt}} M : \sigma \to \tau \qquad \Gamma \vdash_{\mathsf{lt}} N : \sigma}{\Gamma \vdash_{\mathsf{lt}} M\, N : \tau}$	

We can consider two new relations now, $\vdash_{\mathsf{lt}}$ and (Γ valid and $\vdash_{\mathsf{lt}}$).

Notation 5 *For relation* $\vdash_{\mathsf{x}}$*, write* $\Gamma \vdash^{l}_{\mathsf{x}} M : \sigma$ *for* (Γ valid *and* $\Gamma \vdash_{\mathsf{x}} M : \sigma$)*, the* local *version of* $\vdash_{\mathsf{x}}$

Clearly

$$\vdash \quad = \quad \vdash^{l}_{\mathsf{i}} \quad \subset \quad \vdash^{l}_{\mathsf{lt}} \quad \subset \quad \vdash_{\mathsf{lt}}$$

where all the containments are proper, as suggested by the following examples

$$\begin{array}{c} y{:}\varsigma, x{:}\tau \vdash ([z{:}\tau][w{:}\xi]w)\, x : \xi \to \xi \\ y{:}\varsigma, x{:}\tau \vdash^{l}_{\mathsf{lt}} ([x{:}\tau][x{:}\xi]x)\, x : \xi \to \xi \\ x{:}\varsigma, x{:}\tau \vdash_{\mathsf{lt}} ([x{:}\tau][x{:}\xi]x)\, x : \xi \to \xi \end{array}$$

Notice that $\Gamma \vdash_{\mathsf{lt}} M : \sigma$ iff $\vdash_{\mathsf{lt}} \Gamma M : \sigma$ (where by if $\Gamma = x{:}\sigma, y{:}\tau, \ldots$, then $\Gamma M = [x{:}\sigma][y{:}\tau]\ldots M$), while $\vdash^{l}_{\mathsf{lt}}$ doesn't have this property, which is why $\vdash^{l}_{\mathsf{lt}}$ is called the local system of liberal terms.

With $\vdash_{\mathsf{lt}}$ and $\vdash^{l}_{\mathsf{lt}}$ we have systems for typing $\lambda\!\to$ which are closed under alpha-conversion and require no notion of substitution. (But of course we are also interested in reduction and conversion on the typed terms, and these require substitution.)

A criticism of $\vdash_{\mathsf{lt}}$ As I suggested above, the non-operational abstraction "$x \in \Gamma$" that requires x is bound at most once in a valid context, is not well matched to our goal of presenting the typing relation, for the informal notion of term allows the same variable name to be bound more than once, and implicitly contains the idea of "linearly" searching fram a variable instance through enclosing scopes to find the one binding that variable instance.

In fact, presentations of type systems in the style of our presentation of $\vdash$, where validity of a context means any variable name is bound at most once, are very common in the literature (for example [Bar92, Luo90, HHP92]). Why do type theory designers not use presentations in the style of $\vdash_{\mathsf{lt}}$? The problem is that $\vdash_{\mathsf{lt}}$ has bad properties of weakening. If $\Gamma \vdash M : \sigma$ and Γ' contains all the bindings of Γ, and is also valid, then $\Gamma' \vdash M : \sigma$ but $\vdash_{\mathsf{lt}}$ doesn't have this property. This is a logical property which shows that global bindings should not be treated the same as local bindings. Both $\vdash$ and $\vdash_{\mathsf{lt}}$ treat local and global bindings uniformly: $\vdash$ is unsatisfactory because it is too restrictive with local bindings, so is not closed under alpha-conversion; $\vdash_{\mathsf{lt}}$ is unsatisfactory because it is too liberal with global bindings, so is not closed under weakening. Is $\vdash^{l}_{\mathsf{lt}}$ just right?

2.3 $\lambda\!\to$ with nameless variables

A well-known technique to avoid questions of variable names is the use of de Bruijn nameless variables.

Pure nameless terms Here is a presentation of $\lambda\!\to$ for pure nameless (de Bruijn) terms.

DB-START $\quad \Gamma, \sigma \vdash_{\mathsf{db}} 0 : \sigma$

DB-WEAK $\quad \dfrac{\Gamma \vdash_{\mathsf{db}} n : \sigma}{\Gamma, \tau \vdash_{\mathsf{db}} n' : \sigma}$

DB-LDA $\quad \dfrac{\Gamma, \sigma \vdash_{\mathsf{db}} M : \tau}{\Gamma \vdash_{\mathsf{db}} [\sigma]M : \sigma \to \tau}$

DB-APP $\quad \dfrac{\Gamma \vdash_{\mathsf{db}} M : \sigma \to \tau \qquad \Gamma \vdash_{\mathsf{db}} N : \sigma}{\Gamma \vdash_{\mathsf{db}} M\,N : \tau}$

Notice that there are no real choices to be made: there are no restrictions on the context, and we "search" it linearly.

locally nameless terms Now consider terms whose local binding is by de Bruijn indexes, but whose global binding is by named variables. As before, x, y range over a class of variables that will be used for global, or free, variables. As usual, we define two operations of "substitution" (see [Hue89])

$M[N/k]$ replaces the k^{th} *free* index with appropriately lifted instances of N, and lowers all free indexes higher than k since there is no longer a "hole" at k.

$M[k/x]$ replaces name x with the k^{th} *free* index, lifting indexes greater or equal to k to make room for a new free index.

Here is a system for $\lambda\!\to$ typing of locally nameless terms.

LN-VAR $\quad \dfrac{\Gamma \ \mathsf{valid}}{\Gamma \vdash_{\mathsf{ln}} x : \sigma} \qquad x{:}\sigma \in \Gamma$

LN-LDA $\quad \dfrac{\Gamma, x{:}\sigma \vdash_{\mathsf{ln}} M[x/0] : \tau}{\Gamma \vdash_{\mathsf{ln}} [\sigma]M : \sigma \to \tau} \qquad x \notin M$

LN-APP $\quad \dfrac{\Gamma \vdash_{\mathsf{ln}} M : \sigma \to \tau \qquad \Gamma \vdash_{\mathsf{ln}} N : \sigma}{\Gamma \vdash_{\mathsf{ln}} M\,N : \tau}$

This system is very similar in spirit to $\vdash_{\mathsf{s}}$ of section 2.1. Its handling of global names is identical to that of $\vdash_{\mathsf{s}}$ (and $\vdash$), and its central feature, the analysis of how a local variable becomes global, is very reminiscint of $\vdash_{\mathsf{s}}$. Of course $\vdash_{\mathsf{ln}}$ is closed under alpha-conversion, because alpha-conversion and identity are the same for locally nameless terms. The Constructive Engine uses $\vdash_{\mathsf{ln}}$ as the "kernel" of a system for typing conventional named terms that inherits closure under alpha-conversion from $\vdash_{\mathsf{ln}}$.

Remark 6 $\vdash_{\mathsf{db}}$ *is essentially the same as* $\vdash_{\mathsf{lt}}$ *in some way not yet made clear, and similarly* $\vdash_{\mathsf{ln}}$ *is essentially the same as* $\vdash^{l}_{\mathsf{lt}}$*. To make sense of this we should give translations back and forth between named and nameless terms, and show that typing on the named terms and their translations coincides. I have no time to do this at the present writing.*

2.4 A Constructive Engine for $\lambda\!\rightarrow$

Since $\vdash_{\mathsf{ln}}$ treats global names just as $\vdash$ does, we may use the optimization and transformations of section 2.2 on $\vdash_{\mathsf{ln}}$. Analogous to $\vdash_{\mathsf{i}}$ we have $\vdash_{\mathsf{iln}}$

$$\text{ILN-START} \qquad \Gamma, x{:}\sigma \vdash_{\mathsf{iln}} x : \sigma$$

$$\text{ILN-WEAK} \qquad \frac{\Gamma \vdash_{\mathsf{iln}} x : \sigma}{\Gamma, y{:}\tau \vdash_{\mathsf{iln}} x : \sigma} \qquad x \neq y$$

$$\text{ILN-LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash_{\mathsf{iln}} M[x/0] : \tau}{\Gamma \vdash_{\mathsf{iln}} [\sigma]M : \sigma \rightarrow \tau} \qquad x \notin M,\ x \notin \mathsf{Dom}\,(\Gamma)$$

$$\text{ILN-APP} \qquad \frac{\Gamma \vdash_{\mathsf{iln}} M : \sigma \rightarrow \tau \qquad \Gamma \vdash_{\mathsf{iln}} N : \sigma}{\Gamma \vdash_{\mathsf{iln}} M\,N : \tau}$$

Similar to lemmas 3 and 4,

$$\Gamma \vdash_{\mathsf{ln}} M : \sigma \quad \Leftrightarrow \quad (\Gamma \text{ valid and } \Gamma \vdash_{\mathsf{iln}} M : \sigma)$$

Again as in section 2.2 we define a system of "liberal terms" by replacing ILN-LDA with

$$\text{LTLN-LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash_{\mathsf{ltln}} M[x/0] : \tau}{\Gamma \vdash_{\mathsf{ltln}} [\sigma]M : \sigma \rightarrow \tau} \qquad x \notin M$$

In section 2.2 the step from $\vdash_{\mathsf{i}}$ to $\vdash_{\mathsf{lt}}$ changed the derivable judgements; in fact $\vdash_{\mathsf{lt}}$ is closed under alpha-conversion, while $\vdash_{\mathsf{i}}$ is not. In the present case $\vdash_{\mathsf{iln}}$ is already closed under alpha-conversion, so the step to $\vdash_{\mathsf{ltln}}$ does not change the derivable judgements. The main point is that x occurs in the conclusion of I-LDA but not in the conclusion of ILN-LDA.

Lemma 7 (Correctness of $\vdash_{\mathsf{ltln}}$)
$\Gamma \vdash_{\mathsf{ln}} M : \sigma \quad \Leftrightarrow \quad \Gamma \vdash_{\mathsf{ltln}} M : \sigma$

Proof It suffices to show $\Gamma \vdash_{\mathsf{iln}} M : \sigma \;\Leftrightarrow\; \Gamma \vdash_{\mathsf{ltln}} M : \sigma$. Direction $\Rightarrow$ is trivial. Prove direction $\Leftarrow$ by induction on a derivation of $\Gamma \vdash_{\mathsf{ltln}} M : \sigma$. For the case LTLN-LDA, if $x \in \mathsf{Dom}\,(\Gamma)$, just choose another x. ∎

The Constructive Engine We are almost ready to present the Constructive Engine for $\lambda\!\rightarrow$. It is (a system of rules for) an inductive relation of the shape $\Gamma \vdash M \Rightarrow \overline{M} : \sigma$. Γ and M are concrete objects with named variables, which we think of as inputs to the engine. $\overline{M}$ and σ are the outputs, respectively the translation of M into locally nameless form, and the Γ-type of $\overline{M}$ in $\vdash_{\mathsf{ltln}}$. For example, the rule for application terms is

$$\text{CE-APP} \qquad \frac{\Gamma \vdash M \Rightarrow \overline{M} : \sigma \rightarrow \tau \qquad \Gamma \vdash N \Rightarrow \overline{N} : \sigma}{\Gamma \vdash M\,N \Rightarrow \overline{M}\,\overline{N} : \tau}$$

This is read "to translate and compute a type for the named term $M\,N$ in context Γ (i.e. to evaluate the conclusion of the rule given its inputs), translate and compute types for M and N (i.e. evaluate the premisses of the rule, whose inputs are computed from the given inputs to the conclusion), and return a result computed from the results of the premisses". We have called such systems *translation systems* [Pol90].

There is one difficulty remaining, with the rule for lambda terms. Following the CE-APP example, it should be

$$\frac{\Gamma, x{:}\sigma \vdash M \Rightarrow \overline{M}[x/0] : \tau}{\Gamma \vdash [x{:}\sigma]M \Rightarrow [\sigma]\overline{M} : \sigma \to \tau} \qquad x \notin \overline{M}$$

but it is not clear how to compute $[\sigma]\overline{M}$ from $\overline{M}[x/0]$. Reading LTLN-LDA algorithmicly the term $[\sigma]M$ is the input: to compute its type, strip off the lambda, put a variable x in the hole thus created, (i.e. $M[x/0]$) and compute a type for this in an extended context. In the translation system this is dualized: given the named term $[x{:}\sigma]M$, translate the named term M to locally nameless term $\overline{M}$, and then somehow construct a locally nameless version of $[x{:}\sigma]M$. To fix this problem, we consider one more system, the same as $\vdash_{\mathsf{ltln}}$ except that LTLN-LDA is replaced by

$$\text{PCE-LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash_{\mathsf{pce}} N : \tau}{\Gamma \vdash_{\mathsf{pce}} [\sigma](N[0/x]) : \sigma \to \tau}$$

(PCE is for *pre-constructive-engine*) and claim:

Lemma 8 (Correctness of $\vdash_{\mathsf{pce}}$)
$\Gamma \vdash_{\mathsf{ltln}} M : \sigma \quad \Leftrightarrow \quad \Gamma \vdash_{\mathsf{pce}} M : \sigma$

Proof First, we have the equations

$$\overline{M}[0/x][x/0] = \overline{M} \qquad (2)$$
$$\overline{M}[x/0][0/x] = \overline{M} \qquad \text{if } x \notin \overline{M} \qquad (3)$$

Direction $\Leftarrow$ is easy, for if a PCE-derivation ends with

$$\text{PCE-LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash_{\mathsf{pce}} N : \tau}{\Gamma \vdash_{\mathsf{pce}} [\sigma](N[0/x]) : \sigma \to \tau}$$

just apply LTLN-LDA with $M = N[0/x]$, using equation (2) and the fact that $x \notin N[0/x]$ no matter what N is.

Conversely, assume a LTLN-derivation ends with

$$\text{LTLN-LDA} \qquad \frac{\Gamma, x{:}\sigma \vdash_{\mathsf{ltln}} M[x/0] : \tau}{\Gamma \vdash_{\mathsf{ltln}} [\sigma]M : \sigma \to \tau} \qquad x \notin M$$

Since $x \notin M$, using equation (3), apply PCE-LDA with $N = M[x/0]$. ■

Now we can give the Constructive Engine for $\lambda\!\to$:

AX	$\bullet \vdash s_1 : s_2$	$\mathsf{Ax}(s_1{:}s_2)$
START	$\dfrac{\Gamma \vdash A : s}{\Gamma[x{:}A] \vdash x : A}$	$x \notin \mathsf{Dom}\,(\Gamma)$
WEAK	$\dfrac{\Gamma \vdash \sigma : C \qquad \Gamma \vdash A : s}{\Gamma[x{:}A] \vdash \sigma : C}$	σ is a sort or a variable, $x \notin \mathsf{Dom}\,(\Gamma)$
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	$\dfrac{\Gamma \vdash M : \{x{:}A\}B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B[N/x]}$	
CONV	$\dfrac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \simeq B}{\Gamma \vdash M : B}$	

Table 3: The typing judgement of a PTS.

CE-START	$\Gamma, x{:}\sigma \vdash x \Rightarrow x : \sigma$	
CE-WEAK	$\dfrac{\Gamma \vdash x \Rightarrow \overline{x} : \sigma}{\Gamma, y{:}\tau \vdash x \Rightarrow \overline{x} : \sigma}$	$x \neq y$
CE-LDA	$\dfrac{\Gamma, x{:}\sigma \vdash M \Rightarrow \overline{M} : \tau}{\Gamma \vdash [x{:}\sigma]M \Rightarrow [\sigma](\overline{M}[0/x]) : \sigma \to \tau}$	
CE-APP	$\dfrac{\Gamma \vdash M \Rightarrow \overline{M} : \sigma \to \tau \qquad \Gamma \vdash N \Rightarrow \overline{N} : \sigma}{\Gamma \vdash M\,N \Rightarrow \overline{M}\,\overline{N} : \tau}$	

This system has a clear operational reading. Given a derivation of $\Gamma \vdash N \Rightarrow \overline{N} : \sigma$, just erase the named terms to get a derivation of $\Gamma \vdash_{\mathsf{pce}} \overline{N} : \sigma$. If we show that the translation from named terms to locally nameless terms is correct (I will not do so now) the correctness of this engine is established.

One final point: we have the choice of removing the side condition $x \notin \mathsf{Dom}\,(\Gamma)$ from rule CONS-VALID, or not. In the first case we get $\vdash_{\mathsf{lt}}$, in the second case $\vdash^{l}_{\mathsf{lt}}$.

3 Dependent Types

We will work with the familiar class of Pure Type Systems [Bar91, Bar92, GN91, Ber90, MP93, vBJMP93, vBJ93], which, without further ado, we present as the system of rules in Table 3.

A new difficulty arises with alpha-conversion in dependent types: the binding dependency of a term and its type may be different. For example, we expect to be able to derive

$$A{:}*, P{:}A \to * \vdash [x{:}A][x{:}Px]x : \{x{:}A\}\{y{:}Px\}Px$$

but not to derive

$$A{:}*, P{:}A \to * \vdash [x{:}A][x{:}Px]x : \{x{:}A\}\{x{:}Px\}Px$$

This example suggests that the rule

$$\textsc{lda} \qquad \frac{\Gamma[x{:}A] \vdash M : B \qquad \Gamma \vdash \{x{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{x{:}A\}B}$$

using the same bound variable for the term and its type is not exactly what we intend.

A formalization of PTS that distinguishes between parameters and variables, along the lines discussed in section 2.1, is described in [MP93]. We use the following LDA rule:

$$\textsc{lda} \qquad \frac{\Gamma, p{:}A \vdash M[p/x] : B[p/y] \qquad \Gamma \vdash \{y{:}A\}B : s}{\Gamma \vdash [x{:}A]M : \{y{:}A\}B} \qquad p \notin M,\ p \notin B$$

I recently proved that this system is closed under alpha-conversion, but the proof is not completely satisfactory, as it uses the rule CONV only for alpha-conversion in several cases. (After all, a presentation of PTS using nameless terms will never use CONV for alpha-conversion.) I hope a better, more intensional, proof can be found, but it is not clear how to do it.

In section 2.2 we derived a system, $\vdash_{\mathsf{lt}}$, for $\lambda\!\to$ without any variable renaming that was closed under alpha-conversion. I don't think we can do the same for PTS. If we follow the transformations of section 2.2, first optimizing to only check context validity once, then linearizing context search, we arrive at the system of Table 4. This system is correct, in the sense

$$\Gamma \vdash M : A \quad \Leftrightarrow \quad (\Gamma \vdash_{\mathsf{vc}} \text{and } \Gamma \vdash_{\mathsf{vc}} M : A)$$

If we now try to drop the side conditions $x \notin \mathsf{Dom}(\Gamma)$ from VC-PI and VC-LDA, as in section 2.2, (call this system $\vdash_{\mathsf{bad}}$) we find the following incorrect derivation

$$\cfrac{\cfrac{\Gamma, x{:}A, x{:}Px \vdash_{\mathsf{bad}} x : Px}{\Gamma, x{:}A \vdash_{\mathsf{bad}} [x{:}Px]x : \{x{:}Px\}Px}}{\Gamma \vdash_{\mathsf{bad}} [x{:}A][x{:}Px]x : \{x{:}A\}\{x{:}Px\}Px}$$

3.1 The Constructive Engine

I remind you that this paper is not addressing the issue of making PTS syntax directed; the Constructive Engine we will derive now is not yet a program for typechecking PTS, but does explain the interaction between named and nameless variables of an operational Constructive Engine.

Table 5 is a correct presentation of PTS using locally nameless terms, corresponding to $\vdash_{\mathsf{iln}}$ of section 2.4. Now, as in section 2.4, we may drop the side condition $x \notin \mathsf{Dom}(\Gamma)$ from rules ILN-PI and ILN-LDA (getting the system $\vdash_{\mathsf{ltln}}$), just as in lemma 7. Here we are using the locally nameless representation in an essential way for dependent types!

Continuing as in section 2.4, we use the argument of lemma 8 to replace LTLN-PI and LTLN-LDA by

VC-SRT	$\Gamma \vdash_{\mathsf{vc}} s_1 : s_2$	$\mathsf{Ax}(s_1{:}s_2)$
VC-VAR	$\Gamma \vdash_{\mathsf{vc}} x : \mathsf{assoc}\ x\ \Gamma$	
VC-PI	$\dfrac{\Gamma \vdash_{\mathsf{vc}} A : s_1 \qquad \Gamma, x{:}A \vdash_{\mathsf{vc}} B : s_2}{\Gamma \vdash_{\mathsf{vc}} \{x{:}A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$ $x \notin \mathsf{Dom}\,(\Gamma)$
VC-LDA	$\dfrac{\Gamma, x{:}A \vdash_{\mathsf{vc}} b : B \qquad \Gamma \vdash_{\mathsf{vc}} \{x{:}A\}B : s}{\Gamma \vdash_{\mathsf{vc}} [x{:}A]b : \{x{:}A\}B}$	$x \notin \mathsf{Dom}\,(\Gamma)$
VC-APP	$\dfrac{\Gamma \vdash_{\mathsf{vc}} a : \{x{:}B\}A \qquad \Gamma \vdash_{\mathsf{vc}} b : B}{\Gamma \vdash_{\mathsf{vc}} a\ b : A[b/x]}$	
VC-CNV	$\dfrac{\Gamma \vdash_{\mathsf{vc}} a : A \qquad \Gamma \vdash_{\mathsf{vc}} B : s \qquad A \simeq B}{\Gamma \vdash_{\mathsf{vc}} a : B}$	
NIL-VC	$\bullet \vdash_{\mathsf{vc}}$	
CONS-VC	$\dfrac{\Gamma \vdash_{\mathsf{vc}} \qquad \Gamma \vdash_{\mathsf{vc}} A : s}{\Gamma, x{:}A \vdash_{\mathsf{vc}}}$	$x \notin \mathsf{Dom}\,(\Gamma)$

Table 4: The system of valid contexts.

ILN-SRT	$\Gamma \vdash_{\mathsf{iln}} s_1 : s_2$	$\mathsf{Ax}(s_1{:}s_2)$
ILN-VAR	$\Gamma \vdash_{\mathsf{iln}} x : \mathsf{assoc}\ x\ A$	
ILN-PI	$\dfrac{\Gamma \vdash_{\mathsf{iln}} A : s_1 \qquad \Gamma, x{:}A \vdash_{\mathsf{iln}} B[x/0] : s_2}{\Gamma \vdash_{\mathsf{iln}} \{A\}B : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$ $x \notin B,\ \ x \notin \mathsf{Dom}\,(\Gamma)$
ILN-LDA	$\dfrac{\Gamma, x{:}A \vdash_{\mathsf{iln}} b[x/0] : B[x/0] \qquad \Gamma \vdash_{\mathsf{iln}} \{A\}B : s}{\Gamma \vdash_{\mathsf{iln}} [A]b : \{A\}B}$	$x \notin b,\ \ x \notin B$ $x \notin \mathsf{Dom}\,(\Gamma)$
ILN-APP	$\dfrac{\Gamma \vdash_{\mathsf{iln}} a : \{B\}A \qquad \Gamma \vdash_{\mathsf{iln}} b : B}{\Gamma \vdash_{\mathsf{iln}} a\ b : A[b/0]}$	
ILN-CNV	$\dfrac{\Gamma \vdash_{\mathsf{iln}} a : A \qquad \Gamma \vdash_{\mathsf{iln}} B : s \qquad A \simeq B}{\Gamma \vdash_{\mathsf{iln}} a : B}$	
ILN-NIL	$\bullet \vdash_{\mathsf{iln}}$	
ILN-CONS	$\dfrac{\Gamma \vdash_{\mathsf{iln}} \qquad \Gamma \vdash_{\mathsf{iln}} A : s}{\Gamma, x{:}A \vdash_{\mathsf{iln}}}$	$x \notin \mathsf{Dom}\,(\Gamma)$

Table 5: The intermediate system of locally nameless terms.

CE-SRT	$\Gamma \vdash s_1 \Rightarrow s_1 : s_2$	$\mathsf{Ax}(s_1{:}s_2)$
CE-VAR	$\Gamma \vdash x \Rightarrow x : \mathsf{assoc}\ x\ A$	
CE-PI	$\dfrac{\Gamma \vdash A \Rightarrow \overline{A} : s_1 \qquad \Gamma, x{:}A \vdash B \Rightarrow \overline{B} : s_2}{\Gamma \vdash \{x{:}A\}B \Rightarrow \{\overline{A}\}(\overline{B}[0/x]) : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
CE-LDA	$\dfrac{\Gamma, x{:}A \vdash b \Rightarrow \overline{b} : \overline{B} \qquad \Gamma \vdash \{x{:}A\}B \Rightarrow \{\overline{A}\}\overline{B} : s}{\Gamma \vdash [x{:}A]b \Rightarrow [\overline{A}](\overline{b}[0/x]) : \{\overline{A}\}(\overline{B}[0/x])}$	
CE-APP	$\dfrac{\Gamma \vdash a \Rightarrow \overline{a} : \{\overline{B}\}\overline{A} \qquad \Gamma \vdash b \Rightarrow \overline{b} : \overline{B}}{\Gamma \vdash a\ b \Rightarrow \overline{a}\,\overline{b} : \overline{A}[\overline{b}/0]}$	
CE-CNV	$\dfrac{\Gamma \vdash a \Rightarrow \overline{a} : \overline{A} \qquad \Gamma \vdash B \Rightarrow \overline{B} : s \qquad \overline{A} \simeq \overline{B}}{\Gamma \vdash a \Rightarrow \overline{a} : \overline{B}}$	
CE-NIL	$\bullet \vdash_{\mathsf{iln}}$	
CE-CONS	$\dfrac{\Gamma \vdash_{\mathsf{iln}} \qquad \Gamma \vdash A \Rightarrow \overline{A} : s}{\Gamma, x{:}A \vdash_{\mathsf{iln}}}$	$x \notin \mathsf{Dom}\,(\Gamma)$

Table 6: The Constructive Engine for PTS.

PCE-PI	$\dfrac{\Gamma \vdash_{\mathsf{pce}} A : s_1 \qquad \Gamma, x{:}A \vdash_{\mathsf{pce}} B : s_2}{\Gamma \vdash_{\mathsf{pce}} \{A\}(B[0/x]) : s_3}$	$\mathsf{Rule}(s_1, s_2, s_3)$
PCE-LDA	$\dfrac{\Gamma, x{:}A \vdash_{\mathsf{pce}} b : B \qquad \Gamma \vdash_{\mathsf{pce}} \{A\}B : s}{\Gamma \vdash_{\mathsf{pce}} [A](b[0/x]) : \{A\}(B[0/x])}$	

giving a system, $\vdash_{\mathsf{pce}}$, that can be made into a Constructive Engine as in section 2.4. This Constructive Engine (Table 6) is closed under alpha-conversion.

References

[ACCL91] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, March 1993.

[Bar91] Henk Barendregt. Introduction to generalised type systems. *J. Functional Programming*, 1(2):124–154, April 1991.

[Bar92] Henk Barendregt. Lambda calculi 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.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. D. Plotkin, editors, *Logical Frameworks.* Cambridge University Press, 1991.

[dB72] Nicolas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indag. Math.*, 34(5), 1972.

[dB85] Nicolas G. de Bruijn. Generalizing automath by means of a lambda-typed lambda calculus. In *Proceedings of the Maryland 1984-1985 Special Year in Mathematical Logic and Theoretical Computer Science*, 1985.

[DFH+93] Dowek, Felty, Herbelin, Huet, Murthy, Parent, Paulin-Mohring, and Werner. The Coq proof assistant user's guide, version 5.8. Technical report, INRIA-Rocquencourt, February 1993.

[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.

[Gor93] Andrew Gordon. A mechanism of name-carrying syntax up to alpha-conversion. 1993.

[HHP92] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1992. Preliminary version in LICS'87.

[Hue89] Gérard Huet. The constructive engine. In R. Narasimhan, editor, *A Perspective in Theoretical Computer Science.* World Scientific Publishing, 1989. Commemorative Volume for Gift Siromoney.

[Hue93] G. Huet. Residual theory in λ-calculus: A complete Gallina development. 1993.

[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.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In M.Bezem and J.F.Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 289–305. Springer-Verlag, LNCS 664, March 1993.

[Pol90] Robert Pollack. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

[Pol92] R. Pollack. Typechecking in pure type systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, June 1992. available by ftp.

[Tas93] A. Tasistro. Formulation of Martin-Löf's theory of types with explicit substitutions. Master's thesis, Chalmers Tekniska Högskola and Göteborgs Universitet, May 1993.

[vBJ93] L.S. van Benthem Jutting. Typing in pure type systems. *Information and Computation*, 105(1):30–41, July 1993.

[vBJMP93] L. van Benthem Jutting, James McKinna, and Robert Pollack. Typechecking in pure type systems. submitted for publication, 1993.

Machine Deduction

Christophe Raffalli [§]
L.F.C.S. Edinburgh

August 1993

Abstract

We present in this paper a new type system which allows to extract code for an abstract machine instead of lambda-terms. Thus, we get a framework to compile correctly programs extracted from proof by translating their proof in our system and then extracting the code. Moreover, we will see that we can associate programs to classical proofs.

1 Introduction.

The proof as program paradigm, using the Curry-Howard isomorphism [4], gives a way to associate a program to an intuitionistic proof. This program is almost always a functional program (in general a lambda-term [1]) which has to be compiled before being executed [11]. This ensures some correctness about the functional program extracted from the proof. But the correctness of the compiled code is relative to the proof of the compiler.

The usual way to ensure this kind of correctness is to define a semantics for the functional language, and to verify that the compiler preserves this semantics.

We study in this paper a type system for the code of an abstract machine (S.E.C. machine). This approach authorizes a new kind of compilation: we translate the proof in natural deduction to a proof in our system and we extract the code from this new proof.

The two kinds of compilation can be represented by the following diagram:

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{Proof}\\ \text{in}\\ \text{natural deduction}\end{array}} \arrow[r, "\text{Proof translation}"] \arrow[d, "\begin{array}{c}\text{Term}\\ \text{extraction}\end{array}"] & \boxed{\begin{array}{c}\text{Proof}\\ \text{in}\\ \text{M.D.}\end{array}} \arrow[d, "\begin{array}{c}\text{Code}\\ \text{extraction}\end{array}"] \\
\boxed{\text{Lambda-terms}} \arrow[r, "\text{Compilation}"] & \boxed{\text{Code}}
\end{tikzcd}$$

To achieve this goal we define a deduction system MD_{SEC}, for intuitionistic logic. This system is a second order system specially tailored to a translation of Leivant and Krivine's system AF_2[6, 7, 9, 8].

[§] cr@dcs.ed.ac.uk

Secondly, we define an S.E.C. machine and an interpretation for intuitionistic logic in terms of the machine. Then, we show how to extract code from a proof in our system and we prove this code correct for our interpretation: the extracted code belongs to the interpretation of its type.

Next, we show how to deal with data types and optimization, and how the interpretation notion ensures the correctness of the code.

Finally we give one proof translation for call-by-name (in Krivine's style). This means that we can find a proof translation such that the previous diagram commutes for call-by-name compilation.

If the reader is not familiar with second order stuff and system AF_2, he could read this paper forgetting all about first order to use only its propositional part (This restriction of system AF_2 gives the Curry's system F [3]). But doing this, the reader will lose the argument about the correctness of programs in Section 6, because the type characterizes the function in system AF_2 but not in system F.

2 The deduction system.

We use classical second order formulas. First we define first order terms from a language $\mathcal{L}$ defined by an algebraic signature Σ. We choose an infinite set of predicate variables (infinite for each arity) and we construct formulas from atomic formulas, false, true, negation, conjunction, second order and first order existential quantification ($\perp \mid \top \mid X(t_1,\ldots,t_n) \mid F \wedge G \mid \neg F \mid \exists XF \mid \exists xF$). Let $\mathcal{F}$ be this set of formulas.

Definition 2.1 *We choose a partition of second order variables into stack variables ($\mathcal{V}^\Pi$) and value variables ($\mathcal{V}^\Phi$). We define value formulas ($\mathcal{F}^\Phi$) and stack formulas ($\mathcal{F}^\Pi$) as the least subsets of $\mathcal{F}$ verifying:*

$$\begin{array}{rcll}
X(t_1,\ldots,t_n) & \in \mathcal{F}^\Pi & \textit{if} & X \in \mathcal{V}^\Pi \\
A(t_1,\ldots,t_n) & \in \mathcal{F}^\Phi & \textit{if} & A \in \mathcal{V}^\Phi \\
F \wedge P & \in \mathcal{F}^\Pi & \textit{if} & F \in \mathcal{F}^\Phi \;\; \textit{and} \;\; P \in \mathcal{F}^\Pi \\
\exists \chi P & \in \mathcal{F}^\Pi & \textit{if} & P \in \mathcal{F}^\Pi \;\; \textit{and } \chi \textit{ is any kind of variable} \\
\neg P & \in \mathcal{F}^\Phi & \textit{if} & P \in \mathcal{F}^\Pi
\end{array}$$

In all this paper, we will use the following notation to write formulas:

- M, N for any formulas.
- F, G for value formulas.
- P, Q, R for stack formulas.
- X, Y for stack variables.
- A, B for value variables.
- x, y for first order variables.
- t, u for first order terms.

- Γ, Δ for multiset of value formulas.
- $M[t/x]$ for the substitution of the first order variable x with a term t in the formula M.
- $M[\lambda x_1 \ldots \lambda x_n N/X]$ for the substitution of the the second order variable X of arity n with a formula N in the formula M. This substitution is defined as usual, but to be compatible with the definition of stack formulas and value formulas, we will substitute only stack formulas to stack variables and value formulas to value variables.
- χ for any kind of variable (first order, stack or value variables).
- φ, associated to χ, for an expression which is substitutable to χ:
  - φ is a term if χ is a first order variable
  - $\varphi = \lambda x_1 \ldots \lambda x_n P$ where P is a stack formula if χ is a stack variable or arity n
  - $\varphi = \lambda x_1 \ldots \lambda x_n F$ where F is a value formula if χ is a value variable or arity n

We will consider only sequents of the following form, with $F_1, \ldots, F_n \in \mathcal{F}^\Phi$ and $P \in \mathcal{F}^\Pi$:

$$F_1, \ldots, F_n \mid P \vdash$$

In such a sequent, the "," and "|" must be understand as conjunction. So $F_1, \ldots, F_n \mid P \vdash$ means that we get a contradiction from $F_1 \wedge \ldots \wedge F_n \wedge P$.

Here are the rules of the deduction system MD_{SEC}:

$$\frac{}{F_1, \ldots, F_{m-1}, \neg P, F_{m+1}, \ldots, F_n \mid P \vdash}\, Ax^\Gamma \qquad \frac{F_1, \ldots, F_m, \ldots, F_n \mid (F_m \wedge P) \vdash}{F_1, \ldots, F_m, \ldots, F_n \mid P \vdash}\, Co$$

$$\frac{F_1, \ldots, F_n \mid P \vdash}{\Gamma \mid F_1 \wedge \ldots \wedge F_n \wedge P \vdash}\, \wedge_i \qquad \frac{\Gamma \mid \neg Q \wedge P \vdash \quad \Gamma \mid Q \vdash}{\Gamma \mid P \vdash}\, \wedge\neg_e$$

$$\frac{\Gamma \mid \top \vdash}{\Gamma \mid P \vdash}\, \top_e \qquad \frac{\Gamma \mid \neg(\neg P \wedge \top) \wedge P \vdash}{\Gamma \mid P \vdash}\, \top'_e$$

$$\frac{\Gamma \mid P \vdash \qquad \chi \notin \Gamma}{\Gamma \mid \exists \chi\, P \vdash}\, \exists_i \qquad \frac{\Gamma \mid \exists \chi\, P \vdash}{\Gamma \mid P[\varphi/\chi] \vdash}\, \exists_e$$

$$\frac{}{\Gamma \mid \bot \vdash}\, \bot_s$$

Note: the $\top$ connective is not useful when we use existential quantifier. If we replace $\top$ by $\exists X\, X$ the rule $\top_e$ is derivable and the rule $\bot_s$ is still correct. We give a system using $\top$ to have also a complete propositional version.

Proposition 2.2 *We remark that this deduction system is correct for intuitionistic logic. This means that if we prove $\Gamma \mid P \vdash$ in our system then $\Gamma, P \vdash$ is a valid sequent in intuitionistic logic.*

proof: The proof is done by induction on the proof of $\Gamma \mid P \vdash$. In fact, it's sufficient to remark that all rules are correct (the rule $\top'_e$ could seem classical. But because formulas are to the left, this rule is in fact equivalent to $\neg(\neg\neg P \wedge P) \to \neg P$ which is intuitionisticaly true). ♠

3 Programs and machine.

In this section, we define an S.E.C. machine. We define the set of instructions $\mathcal{I}$, the set of programs $\mathcal{P}$, the set of environments $\mathcal{E}$, the set of values $\mathcal{V}$ and the set of stacks $\mathcal{S}$ as the least sets verifying the following conditions:

$$
\begin{array}{rcll}
\mathrm{Jump}_n & \in & \mathcal{I} & \text{if } n \in \mathbb{N} \\
\mathrm{Pop}_n & \in & \mathcal{I} & \text{if } n \in \mathbb{N} \\
\mathrm{Push}[p] & \in & \mathcal{I} & \text{if } p \in \mathcal{P} \\
\mathrm{Push}_n & \in & \mathcal{I} & \text{if } n \in \mathbb{N} \\
\mathrm{Erase} & \in & \mathcal{I} & \\
\mathrm{Stop} & \in & \mathcal{I} & \\
\mathrm{Save} & \in & \mathcal{I} & \\
\mathrm{Rest} & \in & \mathcal{I} & \\
\mathcal{P} & = & \mathcal{I}^{(\mathbb{N})} & (\mathcal{P} \text{ is the set of finite sequences of instructions}) \\
\mathcal{E} & = & \mathcal{V}^{(\mathbb{N})} & (\mathcal{E} \text{ is the set of finite sequences of values}) \\
\mathcal{V} & = & \mathcal{P} \times \mathcal{E} & \\
\mathcal{S} & = & \mathcal{V}^{(\mathbb{N})} & (\mathcal{S} \text{ is the set of finite sequences of values})
\end{array}
$$

We will use the following notation:

- $i;p$ for the concatenation of the instruction i at the beginning of the program p. we won't use the empty program and we will denote i the program using only one instruction i.
- $(\varphi_1, \varphi_2, \ldots, \varphi_{n-1}, \varphi_n)$ for an environment of length n.
- $()$ for an empty environment.
- $\langle p/e \rangle$ for a value with the program p and the environment e.
- $\varphi \cdot \pi$ for the concatenation of the value φ at the beginning of the stack π.
- ε for the empty stack.

Now, we define the transition function "tr" (this is a partial function), from $\mathcal{P} \times \mathcal{E} \times \mathcal{S}$ to itself. We give this definition by the table 1.

For instructions Save and Rest, we use the canonical isomorphism between $\mathcal{S}$ and $\mathcal{E}$ to store a stack in place of an environment.

Definition 3.1 *We define the partial function* $\mathrm{ex}(\varphi, \pi)$ *from* $\mathcal{V} \times \mathcal{S}$ *to* $\mathcal{S}$. *Given a value* $\varphi = \langle p/e \rangle$ *and a stack* π, *let be* $\{S_n\}_{n \in \mathbb{N}}$ *the sequence defined by*

- $S_0 = (p, e, \pi)$
- $S_{n+1} = \mathrm{tr}(S_n)$

Then,$\mathrm{ex}(\varphi, \pi)$ *is defined if the previous sequence is well defined and if exists an integer* N *such that* $S_N = (\mathrm{Stop};p', e', \pi')$. *In this case, we define* $\mathrm{ex}(\varphi, \pi) = \pi'$.

Proposition 3.2 *If* $\mathrm{tr}(p, e, \pi) = (p', e', \pi')$ *then* $\mathrm{ex}(\langle p/e \rangle, \pi)) = \mathrm{ex}(\langle p'/e' \rangle, \pi')$.

<u>*proof*</u>: the proof comes from the definitions of tr and ex. ♠

Table 7: Here is the complete transition table for the S.E.C. machine:

input code output code	input environment output environment	input stack output stack
$\mathrm{Pop}_m;p$ p	e $(\varphi_1,\ldots,\varphi_m)$	$\varphi_1\cdots\varphi_m\cdot\pi$ π
$\mathrm{Push}[p'];p$ p	e e	π $\langle p'/e\rangle\cdot\pi$
$\mathrm{Push}_m;p$ p	$(\varphi_1,\ldots,\varphi_n)$ $(\varphi_1,\ldots,\varphi_n)$	π $\varphi_m\cdot\pi$
$\mathrm{Jump}_m;p$ p'	$(\varphi_1,\ldots,\varphi_{m-1},\langle p'/e'\rangle,\varphi_{m+1},\ldots,\varphi_n)$ e'	π π
$\mathrm{Erase};p$ p	e e	π ε
$\mathrm{Stop};p$ $\mathrm{Stop};p$	e e	π π
$\mathrm{Save};p$ p	e e	π $\langle\mathrm{Rest}/\pi\rangle\cdot\pi$
$\mathrm{Rest};p$ p'	$e=\pi'$ e'	$\langle p'/e'\rangle\cdot\pi$ π'

4 Semantics.

Definition 4.1 *An interpretation σ, is given by:*

- *A mapping $x\mapsto|x|^\sigma$ from first order variables to $\mathcal{V}$.*
- *A mapping $x\mapsto|f|^\sigma$ from function constants of arity n to $\mathcal{V}^n\to\mathcal{V}$.*
- *A mapping $A\mapsto|A|^\sigma$ from program variables of arity n to $\mathcal{V}^n\to\mathcal{P}(\mathcal{V})$.*
- *A mapping $X\mapsto|X|^\sigma$ from stack variables of arity n to $\mathcal{V}^n\to\mathcal{P}(\mathcal{S})$.*
- *An element $|\bot|^\sigma$ of $\mathcal{S}^\sigma$.*

Definition 4.2 *Given $\Phi\in\mathcal{P}(\mathcal{V})$ and $\Pi\in\mathcal{P}(\mathcal{S})$, we define*

$$\Phi\times\Pi=\{\varphi\cdot\pi;\varphi\in\Phi\quad\text{and}\quad\pi\in\Pi\}$$

Definition 4.3 *Given Π in $\mathcal{P}(\mathcal{S})$, we define the set $\overline{\Pi}$ in $\mathcal{P}(\mathcal{V})$ by:*

$$\overline{\Pi}=\{\varphi\in\mathcal{V},\ \ \text{for all}\ \ \pi\in\Pi\ \ \mathrm{ex}(\varphi,\pi)\in|\bot|^\sigma\}$$

We note that $\varphi\in\overline{\Pi}$ implies that $\mathrm{ex}(\varphi,\pi)$ is well defined for all $\pi\in\Pi$.

It is very important to note that the definition of $\overline{\Pi}$ depends of $|\bot|^\sigma$ which is in fact the set of all legal stacks when a program stops.

Given such an interpretation, we define by induction the interpretation for all first order terms, stack formulas and value formulas. The interpretation of a term is an element of $\mathcal{V}$, the interpretation of a stack formula is an element of $\mathcal{P}(\mathcal{S})$ and the interpretation of a value formula is an element of $\mathcal{P}(\mathcal{V})$:

$$\begin{array}{rcl}
|f(t_1,\ldots,t_n)|^\sigma & = & |f|^\sigma(|t_1|^\sigma,\ldots,|t_n|^\sigma) \\
|X(t_1,\ldots,t_n)|^\sigma & = & |X|^\sigma(|t_1|^\sigma,\ldots,|t_n|^\sigma) \\
|A(t_1,\ldots,t_n)|^\sigma & = & |A|^\sigma(|t_1|^\sigma,\ldots,|t_n|^\sigma) \\
|\top|^\sigma & = & \{\varepsilon\} \\
|\neg P|^\sigma & = & \overline{|P|^\sigma} \\
|F \wedge P|^\sigma & = & |F|^\sigma \times |G|^\sigma \\
|\exists X P|^\sigma & = & \bigcup_{\Pi \in \mathcal{V}^n \to \mathcal{P}(\mathcal{S})} |P|^{\sigma[\Pi/X]} \quad (n \text{ is the arity of } X) \\
|\exists A P|^\sigma & = & \bigcup_{\Phi \in \mathcal{V}^n \to \mathcal{P}(\mathcal{V})} |P|^{\sigma[\Phi/A]} \quad (n \text{ is the arity of } A) \\
|\exists x P|^\sigma & = & \bigcup_{\varphi \in \mathcal{V}'} |P|^{\sigma[\varphi/x]}
\end{array}$$

We introduce also the following notation:

- If $\Gamma = F_1,\ldots,F_n$, we will denote $|\Gamma|^\sigma = \{(\varphi_1,\ldots,\varphi_n), \varphi_i \in |F_i|^\sigma\}$.
- For any interpretation σ, we will denote $\sigma[\Phi/\chi]$ for the usual modification of the interpretation of only one variable.
- To deal with second order, we will also denote $|\lambda x_1 \ldots \lambda x_n\, M|^\sigma$ (with $M \in \mathcal{F}^\Phi$ or $M \in \mathcal{F}^\Pi$) for the function defined $\varphi_1,\ldots,\varphi_n \mapsto |M|^{\sigma[\varphi_1/x_1]\ldots[\varphi_n/x_n]}$.

Proposition 4.4 *We have the usual proposition: for all interpretation σ, for all value formula F, and for all stack formula P, we have:*

$$\begin{array}{rcl}
|F[\varphi/\chi]| & = & |F|^{\sigma[|\varphi|^\sigma/\chi]} \\
|P[\varphi/\chi]| & = & |P|^{\sigma[|\varphi|^\sigma/\chi]}
\end{array}$$

<u>proof</u>: The proof is done by induction on the formulas F and P. ♠

5 The type system.

We can also use proof to associate programs to formulas. As in the Krivine's and Leivant's system AF_2, we choose a set $\mathcal{E}$ of equational axioms on first order terms and add a rule for these equations. Here are the rules with their algorithmic contents:

$$\frac{}{\mathrm{Jump}_m : F_1,\ldots,F_{m-1},\neg P,F_{m+1},\ldots,F_n \mid P \vdash}\,Ax^\Gamma \qquad \frac{p : F_1,\ldots,F_n \mid (F_m \wedge P) \vdash}{\mathrm{Push}_m;p : F_1,\ldots,F_n \mid P \vdash}\,Co$$

$$\frac{p : F_1,\ldots,F_n \mid P \vdash}{\mathrm{Pop}_n;p : \Gamma \mid F_1 \wedge \ldots \wedge F_n \wedge P \vdash}\,\wedge_i \qquad \frac{p : \Gamma \mid \neg Q \wedge P \vdash \quad q : \Gamma \mid Q \vdash}{\mathrm{Push}[q];p : \Gamma \mid P \vdash}\,\wedge\!\neg_e$$

$$\frac{p : \Gamma \mid \top \vdash}{\mathrm{Erase};p : \Gamma \mid P \vdash}\,\top_e \qquad \frac{p : \Gamma \mid \neg(\neg P \wedge \top) \wedge P \vdash}{\mathrm{Save};p : \Gamma \mid P \vdash}\,\top'_e$$

$$\frac{p : \Gamma \mid P \vdash \qquad \chi \notin \Gamma}{p : \Gamma \mid \exists \chi\, P \vdash}\ \exists_i \qquad \frac{p : \Gamma \mid \exists \chi\, P \vdash}{p : \Gamma \mid P[\varphi/\chi] \vdash}\ \exists_e$$

$$\frac{}{\text{Stop} : \Gamma \mid \bot \vdash}\ \bot_s \qquad \frac{p : \Gamma \mid P[t/x] \vdash \quad \mathcal{E} \vdash t = u}{p : \Gamma \mid P[u/x] \vdash}\ Eq$$

Proposition 5.1 *Let σ be an interpretation verifying: $\mathcal{E} \vdash t = u$ implies $|t|^\sigma = |u|^\sigma$ for all first order terms. If we prove $p : \Gamma \mid P \vdash$ then for all $e \in |\Gamma|^\sigma$, we have*

$$\langle p/e \rangle \in |\neg P|^\sigma$$

proof: We prove this result by induction on the proof:

- If the last rule is Ax^Γ:

$$\frac{}{\text{Jump}_m : F_1, \ldots, F_{m-1}, \neg P, F_{m+1}, \ldots, F_n \mid P \vdash}\ Ax^\Gamma$$

  Let $e = (\varphi_1, \ldots, \varphi_n) \in \mathcal{V}$ be such that $\varphi_i \in |F_i|^\sigma$. We have $\text{ex}(\langle \text{Jump}_m/e \rangle, \pi) = \text{ex}(\langle p'/e' \rangle, \pi)$ with $\varphi_m = \langle p'/e' \rangle$. So $\langle \text{Jump}_m/e \rangle \in |\neg P|^\sigma$, because $\varphi_m \in |\neg P|^\sigma$.

- If the last rule is Co

$$\frac{p : F_1, \ldots, F_m, \ldots F_n \mid (F_m \wedge P) \vdash}{\text{Push}[m];p : F_1, \ldots, F_m, \ldots F_n \mid P \vdash}\ Co$$

  We denote $\Gamma = F_1, \ldots, F_n$. We have to prove $\langle \text{Push}_m;p/e \rangle \in |\neg P|^\sigma$. We choose $e \in |\Gamma|^\sigma$ and $\pi \in |P|^\sigma$. By definition, we have $e = \langle \varphi_1 \ldots \varphi_m \ldots \varphi_n \rangle$ with for all i, $\varphi_i \in |F_i|^\sigma$. Thus, if we denote $\pi' = \varphi_m \cdot \pi$ we get $\text{ex}(\langle \text{Push}_m;p/e \rangle, \pi) = \text{ex}(\langle p/e \rangle, \pi')$. Hence, we get the expected result using $\langle p/e \rangle \in |\neg(F_m \wedge P)|^\sigma$ (by induction hypothesis) and $\pi' \in |F_m \wedge P|^\sigma$.

- If the last rule is $\wedge_i$

$$\frac{p : F_1, \ldots, F_n \mid P \vdash}{\text{Pop}_n;p : \Gamma \mid F_1 \wedge \ldots \wedge F_n \wedge P \vdash}\ \wedge_i$$

  We denote $\Gamma' = F_1, \ldots, F_n$. We have to prove $\langle \text{Pop}_n;p/e \rangle \in |\neg(F_1 \wedge \ldots \wedge F_n \wedge P)|^\sigma$ for all $e \in |\Gamma|^\sigma$. We choose $e \in |\Gamma|^\sigma$ and $\pi \in |F_1 \wedge \ldots \wedge F_n \wedge P|^\sigma$. By definition, we have $\pi = \varphi_1 \cdots \varphi_n \cdot \pi'$ with for all i, $\varphi_i \in |F_i|^\sigma$ and $\pi' \in |P|^\sigma$. Then, $\text{ex}(\langle \text{Pop}_n;p/e \rangle, \pi) = (\langle p/e' \rangle, \pi')$ with $e' = (\varphi_1, \ldots, \varphi_n)$, and we get the result using $e' \in |\Gamma'|^\sigma$, $\langle p/e' \rangle \in |\neg P|^\sigma$ (induction hypothesis) and $\pi' \in |P|^\sigma$.

- If the last rule is $\wedge\!\neg_e$

$$\frac{p : \Gamma \mid \neg Q \wedge P \vdash \quad q : \Gamma \mid Q \vdash}{\text{Push}[q];p : \Gamma \mid P \vdash}\ \wedge\!\neg_e$$

  We have to prove $\langle \text{Push}[q];p/e \rangle \in |\neg P|^\sigma$. We choose $e \in |\Gamma|^\sigma$ and $\pi \in |P|^\sigma$. We have $\text{ex}(\langle \text{Push}[q];p/e \rangle, \pi) = \text{ex}(\langle p/e \rangle, \langle q/e \rangle \cdot \pi)$. Thus, we get the result because by induction hypothesis we have $\langle p/e \rangle \in |\neg(\neg Q \wedge P)|^\sigma$ and $\langle q/e \rangle \in |\neg Q|^\sigma$, so we have $\langle q/e \rangle \cdot \pi \in |\neg Q \wedge P|^\sigma$.

- If the last rule is $\top_e$

$$\frac{p : \Gamma \mid \top \vdash}{\text{Erase};p : \Gamma \mid P \vdash}\ \top_e$$

Let $e \in |\Gamma|^\sigma$ be, we have to prove $\langle \text{Erase};p/e\rangle \in |\neg P|^\sigma$. By definition, for all $\pi \in |P|^\sigma$, we have $\text{ex}(\langle \text{Erase};p/e\rangle, \pi) = \text{ex}(\langle p/e\rangle, \varepsilon)$. Hence, we get the wanted result because $\langle p/e\rangle \in |\neg\top|^\sigma$ (induction hypothesis).

- If the last rule is $\bot_s$

$$\frac{}{\text{Stop} : \Gamma \mid \bot \vdash}\ \bot$$

For all $e \in |\Gamma|^\sigma$ and $\pi \in |\bot|^\sigma$, we have $\text{ex}(\langle \text{Stop}/e\rangle, \pi) = \pi \in |\bot|^\sigma$. So we have $\langle \text{Stop}/e\rangle \in |\neg\bot|^\sigma$.

- If the last rule is $\top_c$

$$\frac{p : \Gamma \mid \neg(\neg P \wedge \top) \wedge P \vdash}{\text{Save};p : \Gamma \mid P \vdash}\ \top_c$$

Let $e \in |\Gamma|^\sigma$ be, we have to prove that $\langle \text{Save};p/e\rangle \in |\neg P|^\sigma$. Let $\pi \in |P|^\sigma$ be, so we have $\text{ex}(\langle \text{Save};p/e\rangle, \pi) = \text{ex}(\langle p/e\rangle, \varphi \cdot \pi)$, with $\varphi = \langle \text{Rest}/\pi\rangle$. If we prove $\varphi \in |\neg(\neg P \wedge \top)|^\sigma$, we get $\varphi \cdot \pi \in |\neg(\neg P \wedge \top) \wedge P|^\sigma$ and with the induction hypothesis, we get the wanted result.

Now let us prove that $\varphi \in |\neg(\neg P \wedge \top)|^\sigma$. Let us choose $\pi' \in |\neg P \wedge \top|^\sigma$. By definition, $\pi' = \langle p'/e'\rangle \cdot \varepsilon$ with $\langle p'/e'\rangle \in |\neg P|^\sigma$. But we have $\text{ex}(\langle \text{Rest}/\pi\rangle, \pi') = \text{ex}(\langle p'/e'\rangle, \pi)$. Hence we get the expected result because $\langle p'/e'\rangle \in |\neg P|^\sigma$ and $\pi \in |P|^\sigma$.

- If the last rule is $\exists_i$

$$\frac{p : \Gamma \mid P \vdash \qquad \chi \notin \Gamma}{p : \Gamma \mid \exists\chi\, P \vdash}\ \exists_i$$

Let σ be an interpretation and choose $e \in |\Gamma|^\sigma$, we have to prove that $\langle p/e\rangle \in |\neg\exists\chi\, P|^\sigma$. But this signify that for all possible interpretation φ for the variable χ, we have $\langle p/e\rangle \in \overline{|P|^{\sigma[\varphi/\chi]}}$, and this is true by induction hypothesis.

- If the last rule is $\exists_e$

$$\frac{p : \Gamma \mid \exists\chi\, P \vdash Q}{p : \Gamma \mid P[\varphi/\chi] \vdash Q}\ \exists_e$$

By induction hypothesis, for all interpretation σ and for all $e \in |\Gamma|^\sigma$, $\langle p/e\rangle \in |\neg\exists\chi\, P|^\sigma$, so we have $\langle p/e\rangle \in \overline{|P|^{\sigma[|\varphi|^\sigma/\chi]}}$, for all expression φ substitutable to χ, and by the proposition 4.4, we have $|P[\varphi/\chi]|^\sigma = |P|^{\sigma[|\varphi|^\sigma/\chi]}$. Hence we get $\langle p/e\rangle \in |\neg P[\varphi/\chi]|^\sigma$.

- If the last rule is Eq

$$\frac{p : \Gamma \mid P[t/x] \vdash \quad \mathcal{E} \vdash t = u}{p : \Gamma \mid P[u/x] \vdash}\ Eq$$

By hypothesis on σ and by 4.4, we get $|P[t/x]|^\sigma = |P[u/x]|^\sigma$. So we get the wanted result.

♠

Definition 5.2 *We will say that M is a control formula, if $|M|^\sigma$ is independent of the interpretation σ (we can apply this definition if M is a stack or a value formula). In this case we note $|M|$ the interpretation of M.*

We can see that $\top$ is a control stack formula.

Proposition 5.3 *Let $F_1, \dots, F_n$ be value control formulas and P, Q control stack formulas. If we prove $p : F_1, \dots, F_n \mid \neg Q \wedge P \vdash$, then for all $\pi \in |P|$ and $e \in |F_1, \dots, F_n|$, we have $\mathrm{ex}(\langle p/e \rangle, \varphi \cdot \pi) \in |Q|$, with $\varphi = \langle \mathrm{Stop}/() \rangle$.*

<u>proof</u>: Let $F_1, \dots, F_n$ be value control formulas and P, Q control stack formulas. We assume $p : F_1, \dots, F_n \mid \neg Q \wedge P \vdash$. We can choose an interpretation σ such that $|\bot|^\sigma = |Q|$. So we get $\varphi = \langle \mathrm{Stop}/() \rangle \in |\neg Q|$, and we can apply the proposition 5.1, and we get $\langle p/e \rangle \in |\neg(\neg Q \wedge P)|$ for all $e \in |F_1, \dots, F_n|$. So we have $\mathrm{ex}(\langle p/e \rangle, \varphi \cdot \pi) \in |\bot|^\sigma = |Q|$ for all $\pi \in |P|$. ♠

6 Data types.

It is easy to add data types in this system. Let us show how to add integers. This example is demonstrative enough to show how to do with any kind of data types.

To add Integers, we follow these steps:

- We add a second order value constant of arity one: $N(\_)$. We will use the N symbol without parenthesis to simplify writing. We add the formula Nt to the set of value formulas.
- We add the following symbols to the language: 0, $\mathrm{s}(\_)$, $\mathrm{add}(\_,\_)$, $\mathrm{mul}(\_,\_)$ .... We will use the s symbol without parenthesis to simplify writing. We identify all the elements of the data type with the set of logical terms obtain with its constructors. For integer, this means that the element of the data type N are the logical terms of the form $s^n 0$.
- We add to the set of equations $\mathcal{E}$ some equations to define $\mathrm{add}(\_,\_)$, $\mathrm{mul}(\_,\_)$ ....
- We add all the integers to the set of Value $\mathcal{V}$. So we can put some integers in the the stack or in the environment. We will denote $\bar{i}$ the value associated to an element i of the data type N.
- We add $\mathrm{Push}_{\mathbb{N}}[i]$ (for each integer i), $\mathrm{Rec}_{\mathbb{N}}[p_0][p_s]$ (for each program p_0 and p_s) and $\mathrm{Inc}_{\mathbb{N}}$ to the set of instructions $\mathcal{I}$ and we gives the following transition tables:

input code output code	input environment output environment	input stack output stack
$\mathrm{Push}_{\mathbb{N}}[i];p$ p	e e	π $\bar{i} \cdot \pi$
$\mathrm{Inc}_{\mathbb{N}};p$ p	e e	$\bar{i} \cdot \pi$ $\overline{i+1} \cdot \pi$
$\mathrm{Rec}_{\mathbb{N}}[p_0][p_s]$ p_0	$(\bar{0})$ $()$	π π
$\mathrm{Rec}_{\mathbb{N}}[p_0][p_s]$ p_s	$(\overline{i+1})$ $(\bar{i})$	π $\langle \mathrm{Rec}_{\mathbb{N}}[p_0][p_s]/(\bar{i}) \rangle \cdot \pi$

- We add the following rules to the type system:

$$\frac{p : \Gamma \mid N(i) \wedge P \vdash}{\mathrm{Push}_{\mathbb{N}}[i];p : \Gamma \mid P \vdash}\,\mathbb{N}_{ex} \qquad \frac{p : \Gamma \mid N(\mathrm{s}\,i) \wedge P \vdash}{\mathrm{Inc}_{\mathbb{N}};p : \Gamma \mid N(i) \wedge P \vdash}\,\mathbb{N}_s$$

$$\frac{p_0 : \ \mid P(0) \vdash \qquad p_s : N(j) \mid (\neg P(j) \wedge P(\mathrm{s}\,j)) \vdash}{\mathrm{Rec}_{\mathbb{N}}[p_0][p_s] : N(i) \mid P(i) \vdash}\,\mathbb{N}_r$$

- And finally we extend the notion of interpretation by $|N|^\sigma(i) = \{\bar{i}\}$ if i is an element of the data type N and $|N|^\sigma(u) = \emptyset$ in all the other cases. With this definition it is not difficult to extend the proof of proposition 5.1.

As an example, here is a derivation for the addition program, using the usual equation for addition, $\mathrm{add}(0,j) = j$ and $\mathrm{add}(\mathrm{s}\,i,j) = \mathrm{s}\,\mathrm{add}(i,j)$. To simplify the proof we use this notation:

$$\begin{array}{rcl} P^j(i) & = & \neg(N(\mathrm{add}(i,j)) \wedge Q) \wedge N(j) \wedge Q \\ \Gamma^j(i) & = & \neg P^j(i), \neg(N(\mathrm{add}(\mathrm{s}\,i,j)) \wedge Q) \end{array}$$

Here is the proof:

$$\begin{prooftree}
\AxiomC{}
\RightLabel{Ax^Γ}
\UnaryInfC{$\neg(N\,\mathrm{add}(0,j) \wedge Q) \mid N\,\mathrm{add}(0,j) \wedge Q \vdash$}
\RightLabel{Eq}
\UnaryInfC{$\neg(N\,\mathrm{add}(0,j) \wedge Q) \mid Nj \wedge Q \vdash$}
\RightLabel{$\wedge_i$}
\UnaryInfC{$\mid P^j(0) \vdash$}
\AxiomC{}
\RightLabel{Ax^Γ}
\UnaryInfC{$\Gamma^j(k) \mid P^j(k) \vdash$}
\AxiomC{}
\RightLabel{Ax^Γ}
\UnaryInfC{$\Gamma^j(k) \mid N\,\mathrm{add}(\mathrm{s}\,k,j) \wedge Q \vdash$}
\RightLabel{Eq}
\UnaryInfC{$\Gamma^j(k) \mid N\,\mathrm{s}\,\mathrm{add}(k,j) \wedge Q \vdash$}
\RightLabel{$\mathbb{N}_s$}
\UnaryInfC{$\Gamma^j(k) \mid N\,\mathrm{add}(k,j) \wedge Q \vdash$}
\RightLabel{$\wedge\neg_e$}
\BinaryInfC{$\Gamma^j(k) \mid Nj \wedge Q \vdash$}
\RightLabel{$\wedge_i$}
\UnaryInfC{$Nk \mid \neg P^j(k) \wedge P^j(\mathrm{s}\,k) \vdash$}
\RightLabel{$\mathbb{N}_r$}
\BinaryInfC{$Ni \mid P^j(i) \vdash$}
\RightLabel{Ax^Π}
\UnaryInfC{$Ni, Nj, \neg(N\,\mathrm{add}(i,j) \wedge Q) \mid Nj \wedge Q \vdash$}
\RightLabel{Ax^Π}
\UnaryInfC{$Ni, Nj, \neg(N\,\mathrm{add}(i,j) \wedge Q) \mid Q \vdash$}
\RightLabel{$\wedge_i$}
\UnaryInfC{$\mid Ni \wedge Nj \wedge \neg(N\,\mathrm{add}(i,j) \wedge Q) \wedge Q \vdash$}
\end{prooftree}$$

And the program extracted from this proof:

$$\mathrm{Pop}_3;\mathrm{Push}_2;\mathrm{Push}_3;\mathrm{Rec}_{\mathbb{N}}[\mathrm{Pop}_1;\mathrm{Jump}_1][\mathrm{Pop}_2;\mathrm{Push}[\mathrm{Inc}_{\mathbb{N}};\mathrm{Jump}_2];\mathrm{Jump}_1]$$

We may comment on this proof:

- The conclusion of the proof: $\mid Ni \wedge Nj \wedge \neg(N\,\mathrm{add}(i,j) \wedge Q) \wedge Q \vdash$ is a type for an addition program in continuation passing style. This program waits, on the stack, for two integers of type $N(i)$ and $N(j)$, a "continuation" of type $\neg(N\,\mathrm{add}(i,j) \wedge Q)$ and the rest of the stack of type Q. Then, this program computes the sum of i and j, and calls the continuation .

- We may distinguish three parts in this proof, separated by the recursion rule: The 0 and the successor case and, at the bottom of the proof, some manipulations on the stack to put things in the right order.

7 Correctness and optimizations.

We are going to show how the use of data types as above implies the correctness of the program. In fact we use the fact that by definition data types are control formulas. We can write this proposition:

Proposition 7.1 *If $D_0, D_1, \ldots, D_n$ are some data types, if f is a function symbol of arity n and if we can find a model for our equations $\mathcal{E}$ (this means that our equations don't implies that some distinct elements of a data type are equal). Then if we have a proof of*

$$p : \mid D_1(x_1) \wedge \ldots \wedge D_n(x) \wedge \neg(D_0(f(x_1, \ldots, x_n)) \wedge \top) \wedge \top \vdash$$

the program p is a program for the function f in the sense that for all $i_1, \ldots, i_n$ respectively elements of the data types $D_1, \ldots, D_n$, we have

$$\mathrm{ex}(\langle p/()\rangle\,, \overline{i_1} \cdots \overline{i_n} \cdot \varphi \cdot \varepsilon) = \overline{\Phi(i_1 \cdot \cdots \cdot i_n)} \cdot \varepsilon \quad \textit{with} \quad \varphi = \langle \mathrm{Stop}/()\rangle$$

where Φ is a function from $D_1, \ldots, D_n$ to D_0 verifying for all $i_0, i_1, \ldots, i_n$ in $D_0, D_1, \ldots, D_n$, $\mathcal{E} \vdash \{(\rangle_\infty, \ldots, \rangle_\backslash) = \rangle$, implies $\Phi(i_1, \ldots, i_n) = i_0$.

<u>*proof*</u>: First, we remark that if we can find a model for our equations $\mathcal{E}$, then we can find an interpretation σ such that for all first order terms, $\mathcal{E} \vdash u = v$ implies $|u|^\sigma = |v|^\sigma$. To prove that we use standard method to extend the model defined only on the data to the set of all values.

After this, we may apply the proposition 5.3 to the previous proof (because $D_0, D_1, \ldots, D_n$ and $\top$ are control formula). We get

$$\mathrm{ex}(\langle p/()\rangle\,, \overline{i_1} \cdots \overline{i_n} \cdot \varphi \cdot \pi) \in |D_0(f(x_1, \ldots, x_n)) \wedge Q|^{\sigma[x_1/\overline{i_1}]\ldots[x_n/\overline{i_n}]}$$

Thus by definition of the interpretation σ, we can define the function Φ with $\overline{\Phi(i_1, \ldots, i_n)} \in |D_0(f(i_1, \ldots, i_n))|^\sigma$ and

$$\mathrm{ex}(\langle p/()\rangle\,, \overline{i_1} \cdots \overline{i_n} \cdot \varphi \cdot \varepsilon) = \overline{\Phi(i_1 \cdot \cdots \cdot i_n)} \cdot \varepsilon$$

But, by definition of the interpretation of a data type, there is an unique element in $|D_0(f(i_1, \ldots, i_n))|^\sigma$. Hence we have $|f(i_1, \ldots, i_n)|^\sigma = \overline{\Phi(i_1, \ldots, i_n)}$. So $\mathcal{E} \vdash \{(\rangle_\infty, \ldots, \rangle_\backslash) = \rangle$, implies $\Phi(i_1, \ldots, i_n) = i_0$.

♠

To prove that a program is correct we use only the proposition 5.1 and its corollary 5.3. Then, we can give a very general definition of optimization which preserve these properties.

First we remark that if we add some instructions to the machine (by adding elements to the set $\mathcal{I}$), because the system doesn't use them, all the previous results are preserved. Now, if we consider a derived rule:

$$\frac{p : \Gamma \mid P \vdash Q}{p' : \Gamma' \mid P' \vdash Q'}$$

We may use some new instructions to produce a better programs p''. We preserve the result of the proposition 5.1 using p'' instead of p' if for all interpretation σ, all $e \in |\Gamma|^\sigma$ and all $e' \in |\Gamma'|^\sigma$ we have $\langle p/e\rangle \in |P \to Q|^\sigma$ implies $\langle p''/e'\rangle \in |P' \to Q'|^\sigma$.

In this case we will say that the following rule is an optimized rule:

$$\frac{p : \Gamma \mid P \vdash Q}{p'' : \Gamma' \mid P' \vdash Q'}$$

An important remark is that we don't impose that p' and p'' be "equivalent" programs in any sense. In fact p' and p'' can do totally different things.

This notion of derived rules says that p'' is compatible with the notion of interpretation, this means that it do the same thing than p' only if used with data types in a "well typed environment".

8 Call by name compilation of system AF_2.

The principle of this compilation is to translate a proof from system AF_2 in MD_{SEC} (You can find the definition of system AF_2 in annex A). The first thing is to remark that we can replace the original introduction of implication rule by this multiple introduction:

$$\frac{M_1, \ldots, M_n \vdash_{\mathrm{AF}_2} M}{\Gamma \vdash_{\mathrm{AF}_2} M_1 \to (M_2 \to \ldots (M_n \to M))} \to_i^n$$

The translation of a proof in order to use this rule instead of the original one is in fact the lambda lifting. This first translation is left to the reader. (We need to put a non empty context in the conclusion of the rule because there is no weakening in the system).

Now we translate the formula of system AF_2:

Definition 8.1 *We define by induction on a formula M of system AF_2, a formula $M^0 \in \mathcal{F}^\Pi$, and $\overline{M} \in \mathcal{F}^\Phi$ by*

- $\overline{M} = \neg M^0$
- $X(\overline{t})^0 = X(\overline{t})$ *where* $X \in \mathcal{V}^\Pi$
- $(M \to N)^0 = \overline{M} \wedge N^0$
- $(\forall x\, M)^0 = \exists x\, M^0$
- $(\forall X\, M)^0 = \exists X\, M^0$

Note: All the variables are translated in stack variables, so we identify the set of variables of system AF_2 to the set of stack variables.

Proposition 8.2 *For all formulas M and N of system AF_2 and all variable X of arity n, we have*

$$(M[\lambda x_1 \ldots \lambda x_n N/X])^0 = M^0[\lambda x_1 \ldots \lambda x_n N^0/X]$$

<u>*proof*</u>: We prove this by induction on the formula M. ♠

Now it is possible to translate the proof (For any environment $\Gamma = M_1, \ldots, M_n$ we will note $\overline{\Gamma} = \overline{M_1}, \ldots, \overline{M_n}$):

Proposition 8.3 *For all environment* Γ *and all formula* M *of system* AF_2*,*

$$\Gamma \vdash_{\mathrm{AF}_2} M \quad \textit{implies} \quad \overline{\Gamma} \mid M^0 \vdash$$

<u>*proof*</u>: We prove this by induction on the proof of $\Gamma \vdash_{\mathrm{AF}_2} M$:

- If the last rule is an axiom: We have $M = M_i$ with $\Gamma = M_1, \ldots, M_n$. Hence, we get the expected result using the Ax^Γ rule, because $\overline{M_i} = \neg M_i^0$.

- If the last rule is the elimination of implication, by induction hypothesis, we get $\Gamma \vdash_{\mathrm{AF}_2} N \to M$ implies $\overline{\Gamma} \mid \neg N^0 \wedge M^0 \vdash$ and $\Gamma \vdash_{\mathrm{AF}_2} N$ implies $\overline{\Gamma} \mid N^0 \vdash$. Hence, we get the expected result using the $\wedge\!\neg_e$ rule.

- If the last rule is the multiple introduction of implication, we have $M = N_1 \to (N_2 \to \ldots (N_p \to N))$ and by induction hypothesis we get $N_1, \ldots, N_p \vdash_{\mathrm{AF}_2} N$ implies $\overline{N_1}, \ldots, \overline{N_p} \mid N^0 \vdash$. Hence, we get the expected result using the $\wedge_i$.

- If the last rule is the elimination of a universal quantifier, we have $M = N[\varphi/\chi]$. By induction hypothesis we get $\Gamma \vdash_{\mathrm{AF}_2} \forall\chi\, N$ implies $\overline{\Gamma} \mid \exists\chi\, N^0 \vdash$. If χ is a first order variable then φ is a term and $M^0 = N^0[\varphi/\chi]$. If χ is a second order variable then we get $M^0 = N^0[\varphi^0/\chi]$ by proposition 8.2. Hence, we have the expected result using the $\exists_e$ rule.

- If the last rule is the introduction of a universal quantifier, we have $M = \forall\chi\, N$ and by induction hypothesis we get $\Gamma \vdash_{\mathrm{AF}_2} N$ implies $\overline{\Gamma} \mid N^0 \vdash$. Moreover, we know that χ is not free in $\overline{\Gamma}$. Hence, we get the expected result using the $\exists_i$ rule.

- If the last rule is the equational rule, the induction hypothesis and the rule *Eq* rule give directly the expected result.

♠

Now, if we compare the term extracted from the proof in natural deduction (with the multiple implication introduction rule) and the code extracted from the translated proof, we obtain the following compilation for the lambda-calculus using super-combinator (we use $\overline{t}$ to denote the code of a term t):

- $x_i \mapsto \mathrm{Jump}_i$ (axiom rules are translated with the Ax^Γ rule)

- $\lambda x_1 \ldots \lambda x_n\, t \mapsto \mathrm{Pop}_n;\overline{t}$ (multiple implication introduction rule are translated with the $\wedge_i$ rule)

- $(t\ u) \mapsto \mathrm{Push}[\overline{u}];\overline{t}$ (implication elimination rule are translated with the $\wedge\!\neg_e$ rule)

- Nothing more, because all other rules of system AF_2 have no algorithmic contents and are translated in non-algorithmic rules of MD_{SEC}.

This sounds like a good call by name compilation. Moreover, one can easely obtain usual optimization. For instance we can translate an elimination of implication on an axiom with the *Co* rule instead of using $\wedge\!\neg_e$ and Ax^Γ rules. So we get a well known optimization which gives $(t\ x_i) \mapsto \mathrm{Push}_i;\overline{t}$ instead of $(tx_i) \mapsto \mathrm{Push}[\mathrm{Jump}_i];\overline{t}$.

Moreover, if we add to system AF_2 the $\bot$ formula with both intuitionist or classical absurdity:

$$\frac{\Gamma \vdash_c t : \bot}{\Gamma \vdash_c (A\ t) : M} \qquad \frac{\Gamma \vdash_c t : (M \to \bot) \to M}{\Gamma \vdash_c (C\ t) : M}$$

We can always translate the proof: we extend the translation of formulas with $\bot^0 = \top$ (we get $((M \to \bot) \to M)^0 = \neg(\neg M^0 \wedge \top) \wedge M^0$. The two previous rules are translated using the $\top_e$ and $\top_c$. This gives the following compilation for the A and C operator: $(A\ t) \mapsto \text{Erase};\overline{t}$ and $(C\ t) \mapsto \text{Save};\overline{t}$. This is a possible compilation for the following reduction rules (in call-by-name) for these operators:

$$\begin{array}{lcl} (A\ t\ t_1 \dots t_n) & \triangleright & t \\ (C\ t\ t_1 \dots t_n) & \triangleright & (t\ \lambda x\ (x\ t_1 \dots t_n)\ t_1 \dots t_n) \end{array}$$

Corollary 8.4 *We have: $\Gamma \vdash_{\mathrm{AF}_2} M$ in system AF_2 with the previous rule added if and only if $\overline{\Gamma} \mid M^0 \vdash$ in MD_{SEC}.*

<u>*proof*</u>: The left-right implication is a consequence of the previous translation. The right-left implication is easy to prove and left to the reader (it's a consequence of the proposition 2.2). ♠

9 Conclusion and further outlook.

This new type system shows how it is possible to use the proof as program paradigm for something really different from the lambda-calculus: some code for an abstract machine.

Moreover, it's a good system to translate second order classical logic into intuitionistic logic with an explicit algorithmic content. In fact, we could see a relation between a kind of A-translation and the call-by-name compilation.

A question is to know how this depends on the type system. I have also design another type system for an S.E.C.D. machine (In fact this is the original type system I have found and the actual S.E.C. version is simply a restriction of this system as the machine is). I have proved this system complete for classical logic ! In fact there is only one rule which give classical logic: a rule to save the dump (like the $\top'_e$ rule in MD_{SEC} is used to save the stack). Hence the relations between classical logic and compilation for an abstract machine seems not so clear.

Another problem is the unsymmetrical character of the system: we can only add instructions to the beginning of a program. Hence, some compilations are not possible. For instance, it's possible to compile call-by-value but you don't get the expected code. (because one of the natural compilation of $(u\ v)$ is $\overline{u};\overline{v};\text{Apply}$, so you need to be able to add instructions to the both sides of $\overline{v}$). It seems not to difficult to write a symmetrical system with both left and right rules to add instructions respectively to the beginning and to the end of a program, and a true cut rule to compose two programs. This kind of system could be powerful enough to really reach most of the compilations for a given machine.

The last possible direction for further work is about optimization. Because we use only the notion of interpration to ensure the correctness, we have more facilities to manipulate the code. Moreover, because we have a lot of information inside the proof we could expect a lot of opimizations which are usually very difficult or impossible to do. One possibility could be to save only the position of the stack when we use this feature only as an exception mechanism (this means when we restore the stack only "inside" the procedure which has saved it). Another possibility could be to optimize a program automaticaly using some physical manipulation. This direction of work is perhaps very promising but is not so easy.

A Definition of system AF_2.

System AF_2 uses classical second order formulas construct with first order terms from a language $\mathcal{L}$, atomic formulas, implication, second order and first order universal quantification $(X(t_1, \ldots, t_n) \mid F \to G \mid \forall X\, F \mid \forall x\, F)$.

It uses also a set of equations on first order terms $\mathcal{E}$. Sequent are of the form $x_1 : F_1, \ldots, x_n : F_n \vdash t : F$ where $x_1, \ldots, x_n$ are lambda-variables and where t is a lambda-term (whose free variables are among $x_1, \ldots, x_n$). This sequent may be proved using the following rules:

- Axiom and equational rules:

$$\frac{}{x : F, \Gamma \vdash x : F}\, Ax \qquad \frac{\Gamma \vdash t : F\,[x/u] \qquad E \vdash u = v}{\Gamma \vdash t : F\,[x/v]}\, Eq$$

- Implication rules:

$$\frac{x : F, \Gamma \vdash t : G}{\Gamma \vdash \lambda x\, t : F \to G} \to_i \qquad \frac{\Gamma \vdash t : F \to G \qquad \Gamma \vdash u : F}{\Gamma \vdash (t)u : G} \to_e$$

- First order abstraction rules:

$$\frac{\Gamma \vdash t : F \qquad \chi \notin \Gamma}{\Gamma \vdash t : \forall \chi F}\, \forall_i \qquad \frac{\Gamma \vdash t : \forall \chi F}{\Gamma \vdash t : F[\varphi/\chi]}\, \forall_e$$

The reader could find more information about system AF_2 in [9], [7] and [6].

References

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

[2] C. Böhm. Alcune proprieta delle forme $\beta\eta$-normali nel λk-calculus. Pubblicazioni 696, Instituto per le Applicazioni del Calcolo, Roma, 1968.

[3] J.-Y. Girard. The system F of variable types: fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[4] W. Howard. The formulae-as-types notion of construction. *To H.B. Curry: Essays on combinatory logic, λ-calculus and formalism*, pages 479–490, 1980.

[5] Jean-Louis Krivine. Opérateurs de mise en mémoire et traduction de Godël. Technical Report 3, Equipe de Logique de Paris 7, December 1989.

[6] Jean-Louis Krivine. *Lambda-Calcul : Types et Modèles.* Etudes et Recherches en Informatique. Masson, 1990. Available soon in english version.

[7] Jean-Louis Krivine and Michel Parigot. Programming with proofs. *Inf. Process. Cybern.*, EIK 26(3):149–167, 1990.

[8] Daniel Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44:51–68, 1986.

[9] Michel Parigot. Programming with proofs: a second order type theory. *Lecture Notes in Computer Science*, 300, 1988. Communication at ESOP 88.

[10] Michel Parigot. Recursive programming with proofs. *Theoritical Computer Science*, 94:335–356, 1992.

[11] Simon L. Peyton J. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987. Prentice-Hall International Series in Computer Science.

Type theory and the informal language of mathematics

Aarne Ranta
Department of Philosophy, P.O.Box 24,
00014 University of Helsinki, Finland.

September 1993.

In the first comprehensive formalization of mathematics, the *Begriffsschrift* (1879), Frege gave up the structure of informal language, in order to reveal the structure of mathematical thought itself. Attempts to apply Frege's formalism to informal discourse outside mathematics followed in this century, e.g. by Russell, Carnap, Quine, and Davidson. In this tradition, the application of logical formalism to informal language is an exercise of skill, rather than an algorithmic procedure, precisely because the linguistic structure is different from the logical structure.

It was Chomsky (1957) who started the study of natural language itself as a formal system, inductively defined by the clauses of a generative grammar. But the structure he gave to his fragment of English was quite different from the structure of a logical formalism.

Finally, Montague (1970) unified the enterprises of Frege and Chomsky in an attempt to give a systematic logical formalization to a fragment of English. His grammar applies to a piece of informal discourse outside mathematics. But as modern logic, even in the form employed by Montague, stems from Frege, who designed it for mathematics, a grammar like Montague's should be applicable, if at all, to the language of mathematics.

Following roughly the format of Montague grammar, I have been working within the constructive type theory of Martin-Löf. (See Martin-Löf 1984 and Nordström & al. 1990 for the type theory, and Ranta 1991 and 1993 for the grammar.) Type theory has proved to be structurally closer to natural language than predicate calculus at least at the following points.

First, type theory makes a distinction between substantival and adjectival terms, e.g. between *number* and *prime*. These are formalized as N : set P : (N)prop, respectively, whereas in predicate calculus they are both formalized as one-place propositional functions.

Second, type theory has quantifier phrases, like *every number*— in type theory,

$$\Pi(N) \; : \; ((N)\mathrm{prop})\mathrm{prop}$$

and *every prime number*— in type theory,

$$\Pi(\Sigma(N,P)) \; : \; ((\Sigma(N,P))\mathrm{prop})\mathrm{prop}.$$

Predicate calculus dissolves these quantifier phrases, because it does not have expressions corresponding to them.

Third, type theory has progressive connectives, i.e. a conjunction and an implication of the type

$$(X \; : \; \mathrm{prop})((X)\mathrm{prop})\mathrm{prop}.$$

Such connectives are abundant in informal language, in sentences like *if this equation has a root it is negative.*

To express this in type theory, first look at the implicans *this equation has a root*. It is an existential proposition, of the form

$$\Sigma(R, E).$$

To form the implicandum, use the propositional function *x is negative* defined for $x : R$, i.e.

$$N : (R)\mathrm{prop}$$

in the context of a proof of the implicans,

$$z : \Sigma(R, E).$$

Left projection gives $p(z) : R$, whence

$$N(p(z)) : \mathrm{prop}$$

by application,

$$(z)N(p(z)) : (\Sigma(R, E))\mathrm{prop}$$

by abstraction, and finally

$$\Pi(\Sigma(R, E), (z)N(p(z))) : \mathrm{prop}$$

to express the implication. Predicate calculus, which only has connectives of type

$$(\mathrm{prop})(\mathrm{prop})\mathrm{prop},$$

cannot compose the sentence from the implicans and the implicandum, but must use something like

$$(\forall x)(R(x) \& E(x) \supset N(x)),$$

which does not have constituents formalizing the two subclauses of the sentence in question. This lack of compositionality has been first noted in the discussion of so-called donkey sentences, e.g.

if John owns a donkey he beats it

which has the same form as our mathematical example. Some linguists think such sentences are artificially complicated, but they are certainly abundant in the informal language of mathematics.

1 Formalization and sugaring

There are two directions of grammatical investigation. One can ask:

> How is this sentence / mode of expression / fragment of discourse represented in the formalism?

Questions put in this way, starting with what is given in the informal language, are questions of *formalization*. But one can also start with what is given in the formalism and ask:

> How is this proposition / logical constant / theory expressed in natural language?

These questions will be called questions of *sugaring*.

A special case of formalization is *parsing*: given a string of words belonging to an inductively defined set of such strings, find the grammatical structure. This notion of parsing is of course secondary to the notion of *generation*, the inductive definition of the set of strings. Furthermore, as we can think of generation as the composition of (1) the definition of the formalism and (2) the sugaring of the formalism, we see that parsing is secondary to sugaring in the conceptual order.

2 Basic expressions of geometry

In what follows we shall, even if not define a complete sugaring algorithm, look at mathematics expressed in type theory from the sugaring point of view. We shall apply the sugaring principles of Ranta 1991, 1993, originally presented for everyday discourse (like the donkey sentences), to the language of axiomatic geometry such as in Hilbert 1899 and, within type theory, von Plato 1993.

Start with simple set terms,

$$\mathrm{point} : \mathrm{set},$$
$$\mathrm{line} : \mathrm{set},$$
$$\mathrm{plane} : \mathrm{set}.$$

The sugaring of simple set expressions into common nouns is simple (in the absence of the singular and plural number of nouns),

$$\mathrm{point} \;\; \triangleright \; \textit{point},$$
$$\mathrm{line} \;\; \triangleright \; \textit{line},$$
$$\mathrm{plane} \;\; \triangleright \; \textit{plane}.$$

We use the form $F \;\; \triangleright \;\; E$ to express the relation

$$F \text{ can be sugared into } E.$$

Thus it is not an expression for a clause in a deterministic sugaring algorithm.

Then some propositional functions sugared into verbs and adjectives.

$$\mathrm{lie\_PL} : (\mathrm{point})(\mathrm{line})\mathrm{prop},$$
$$\mathrm{lie\_PL}(a, b) \;\; \triangleright \;\; a \textit{ lies on } b,$$

$$\mathrm{lie\_PPl} : (\mathrm{point})(\mathrm{plane})\mathrm{prop},$$
$$\mathrm{lie\_PPl}(a, b) \;\; \triangleright \;\; a \textit{ lies in } b,$$

$$\mathrm{lie\_LPl} : (\mathrm{line})(\mathrm{plane})\mathrm{prop},$$
$$\mathrm{lie\_LPl}(a, b) \;\; \triangleright \;\; a \textit{ lies in } b,$$

$$\mathrm{parallel} : (\mathrm{line})(\mathrm{line})\mathrm{prop},$$
$$\mathrm{parallel}(a, b) \;\; \triangleright \;\; a \textit{ is parallel to } b,$$

$$\mathrm{equal} : (A : \mathrm{set})(A)(A)\mathrm{prop},$$
$$\mathrm{equal}(A, a, b) \;\; \triangleright \;\; a \textit{ is equal to } b.$$

Observe how sugaring overloads the English expressions *lies in* and *equal*. The adjective *equal* is fully polymorphic, the verb *lies in* has two uses. The adjective *parallel* and the verb *lies on* are, in this fragment at least, uniquely typed.

3 Logical constants

There are quantifier words like

$$\begin{aligned}&\text{every} : (X : \text{set})((X)\text{prop})\text{prop},\\ &\text{Indef} : (X : \text{set})((X)\text{prop})\text{prop},\\ &\text{some} : (X : \text{set})((X)\text{prop})\text{prop}.\end{aligned}$$

A quantified proposition is sugared by replacing the bound variable by a quantifier phrase,

$$\begin{aligned}&\text{every}(A,(x)B) \ \triangleright\ B[\textit{every } A/x],\\ &\text{Indef}(A,(x)B) \ \triangleright\ B[\text{INDART}(A)\ A/x],\\ &\text{some}(A,(x)B) \ \triangleright\ B[\textit{some } A/x].\end{aligned}$$

INDART(A) is the indefinite article corresponding to the sugaring of A, either *a* or *an*. Observe that if the number of occurrences of x in B is other than one, we may get odd results like

$$\begin{aligned}&\text{every}(\text{line},(x)\text{parallel}(x,x))\\ &\quad\triangleright\ \textit{every line is parallel to every line}.\end{aligned}$$

The uniqueness of replacements can be attained e.g. by using pronouns (see Section 6). It is one of the central problems of the logical formalization of natural language, stemming from the apparently quite different modes of expression of quantification in them. Following Frege (1879, § 9), we shall use the word *main argument* for the occurrence of x to be replaced by the quantifier phrase. (See Ranta 1991, Section 5, for a definition of the main argument.)

Another difficulty with the replacement procedure in the sugaring of quantifiers is that the relative scopes of the quantifiers get lost. The rules give e.g.

$$\left.\begin{array}{l}\text{some}(\text{point},(x)\text{every}(\text{line},(y)\text{lie\_PL}(x,y)))\\ \text{every}(\text{line},(y)\text{some}(\text{point},(x)\text{lie\_PL}(x,y)))\end{array}\right\}\ \triangleright\ \textit{some point lies on every line},$$

and such sentences are indeed considered ambiguous in Montague grammar. But it seems that the mathematician would without hesitation interpret the sentence as the first proposition, although it is a plainly false proposition. He would follow the principle according to which the scopes of the quantifiers get narrower from left to right. (On this rule of precedence, as well as some other ones, cf. Ranta 1993, Chapters 3 and 9.)

As for connectives, we introduce two progressive ones and one that is not progressive.

$$\begin{aligned}&\text{if} : (X : \text{prop})((X)\text{prop})\text{prop},\\ &\text{if}(A,(x)B) \ \triangleright\ \textit{if } A, B[\emptyset/x],\\[4pt] &\text{and} : (X : \text{prop})((X)\text{prop})\text{prop},\\ &\text{and}(A,(x)B) \ \triangleright\ A \textit{ and } B[\emptyset/x],\\[4pt] &\text{or} : (\text{prop})(\text{prop})\text{prop},\\ &\text{or}(A,B) \ \triangleright\ A \textit{ or } B,\end{aligned}$$

where $\emptyset$ is the ellipsis, the empty morph.

Connective and quantifier words are not type-theoretical primitives, but have the definitions

$$
\begin{aligned}
&\text{every} = \Pi : (X : \text{set})((X)\text{prop})\text{prop},\\
&\text{Indef} = \Sigma : (X : \text{set})((X)\text{prop})\text{prop},\\
&\text{some} = \Sigma : (X : \text{set})((X)\text{prop})\text{prop},\\
&\text{if} = \Pi : (X : \text{prop})((X)\text{prop})\text{prop},\\
&\text{and} = \Sigma : (X : \text{prop})((X)\text{prop})\text{prop},\\
&\text{or} = + : (\text{prop})(\text{prop})\text{prop}.
\end{aligned}
$$

The main difference between quantifiers and progressive connectives is in the sugaring of the first argument: for quantifiers, it is a common noun, and for connectives, a sentence. But there is another difference, which has to do with the expressive capacities of the two modes of expression in English. We noted before that the sugaring of a quantified proposition $Q(A,(x)B)$ requires there to be precisely one main argument occurrence of x in B. For connectives, there is no such restriction. Thus for instance the vacuous quantification

$$\Pi(\text{equal}(N,0,1),(x)\text{equal}(N,1,10000))$$

gives, by the sugaring rule for every, the falsity *one is equal to ten thousand*, and it is the rule for if that gives the right true proposition,

if zero is equal to one, one is equal to ten thousand.

Thus connectives provide a more widely applicable means of expressing propositions than quantifiers.

4 Objects and expressions

Sugaring is not a function on type-theoretical objects, such as propositions, but on expressions for those objects. For by the extensionality of functions, a proposition would be then sugared in the same way, in whatever way expressed. But we certainly want to sugar every(A,B) differently from if(A,B), although they are both equal to $\Pi(A,B)$. Even more clearly, if we introduce an abbreviatory expression by explicit definition, we want to sugar the definiendum differently from the much longer definiens. Consider, for instance,

$$\text{triangle} = \Sigma(\text{line},(x)\Sigma(\text{point},(y)\text{outside\_PEl}(y,\text{extended}(x)))) : \text{set},$$

where outside_PEl(a,b) says that the point a lies outside the extended line b, and extended(a) is the infinite extension of the finite line a.

In general, we want to introduce so many definitional variants of type-theoretical expressions that there is a one-to-one correspondence between English and type-theoretical expressions.

The propositions as types principle is, analogously, assumed for the objects of type theory only. We want the type prop to correspond to sentences, and the type set to common nouns. For type-theoretical objects. we have

$$\text{prop} = \text{set} : \text{type},$$

but for expressions, this equation is not effective, whereas we assume the transformation

$$\text{there} = (X)X : (\text{set})\text{prop}$$

sugared

$$\text{there}(A) \;\; \triangleright \;\; \textit{there is}\ \text{INDART}(A)\ A.$$

It may happen that some expression cannot be sugared, e.g. if it contains a quantifier with no or multiple main arguments. In such a case, the sugaring of the proposition expressed must proceed by finding a definitional variant that can be sugared.

5 Relative pronouns

To form complex set terms, we can use relative pronouns, e.g.

$$\begin{aligned}&\text{that} = \Sigma : (X : \text{set})((X)\text{prop})\text{set},\\ &\text{that}(A, (x)B) \;\; \triangleright \;\; A\ \textit{that}\ B[\emptyset/x],\end{aligned}$$

$$\begin{aligned}&\text{such\_that} = \Sigma : (X : \text{set})((X)\text{prop})\text{set},\\ &\text{such\_that}(A, (x)B) \;\; \triangleright \;\; A\ \textit{such that}\ B[\emptyset/x].\end{aligned}$$

These definitions accord with Martin-Löf's (1984) explanation of *such that* as forming a set of elements of the basic set paired with witnessing information. This treatment is necessary for a compositional formalization of quantifier phrases whose domains are given by using relative clauses, and reference is also made to the witnessing information; cf. Section 7 below. But at the same time, we will have to sugar e.g.

$$\text{every}(\text{that}(A, B), (x)C) \;\; \triangleright \;\; C[\text{that}(A, B)/p(x)],$$

i.e. not replace x but $p(x)$. This can be accomplished by the general rule

$$p(x) \;\; \triangleright \;\; x.$$

The slight unnaturalness of the solution is, so it seems to me, one instance of the problems we have in formalizing separated subsets by Σ and trying to get rid of the extra information in some cases, while having to keep it in some other cases.

The difference between that and such_that is analogous to the difference between quantifiers and connectives: that requires there to be exactly one main argument in the relative clause, but such_that does not. such_that is thus more widely applicable than that.

6 Anaphoric expressions

Pronouns are introduced to our fragment of English by the rules

$$\begin{aligned}&\text{Pron} = (X)(x)x : (X : \text{set})(X)X,\\ &\text{Pron}(A, a) \;\; \triangleright \;\; \text{PRO}(A).\end{aligned}$$

In mathematical language, we do not need *he* or *she*, so $\text{PRO}(A)$ is always *it*, and we could as well have

$$\begin{aligned}&\text{it} = (X)(x)x : (X : \text{set})(X)X,\\ &\text{it}(A, a) \;\; \triangleright \;\; \textit{it}.\end{aligned}$$

Observe that our definition of *it* as the polymorphic identity mapping whose argument is sugared away is very similar to the *it* of ML. The main difference is the *interpretation rule* stating in what situations *it* may replace a singular term. In ML, *it* always refers to the value of the latest value declaration. But this rule is too simple for the informal language of mathematics, where *it* can have different—yet definite—interpretations in one and the same clause, e.g.

if the function f has a maximum, it reaches it at least twice.

Our main rule regulating the use of pronouns (and other anaphoric expressions; see below) is that

> the interpretation of an anaphoric expression is an object of appropriate type given in the context in which the expression is used.

Context here is, in the technical sense of type theory, a list of declarations of variables assumed when the expression is formed. For instance, the proposition B in $\Pi(A,(x)B)$ is formed in the context $x : A$. To these variables we add the constant singular terms used in the same sentence. Moreover, we close the "universe of discourse" based on the context under selector operations (cf. Ranta 1993, Chapter 4, for more details).

An interpretation $a : A$ of a pronoun E in the English expression $---E---$ must thus fulfil the following two conditions.

$$\mathrm{Pron}(A,a) \ \rhd\ E,$$

there is a propositional function $B(x) : \mathrm{prop}\ (x : A)$
such that $B(a) \ \rhd\ ---a---.$

As the only pronoun in the mathematical fragment is *it*, the first condition is always satisfied. If there are many objects given in context, it is the second condition that saves the *uniqueness* of reference, expressed by the principle that

> the interpretation of an anaphoric expression must be unique in the context in which it is used.

There are other anaphoric expressions besides pronouns, more specific in the sense that they do not suppress all information about the object referred to. A definite noun phrase formed by the definite article the preserves the type of the object. A modified definite phrase formed by Mod makes explicit some more information given about the object in the context.

$$\mathrm{the} = (X)(x)x : (X : \mathrm{set})(X)X,$$
$$\mathrm{the}(A,a) \ \rhd\ \mathit{the}\,A.$$
$$\mathrm{Mod} = (X)(Y)(x)(y)x : (X : \mathrm{set})(Y : (X)\mathrm{prop})(x : X)(Y(x))X,$$
$$\mathrm{Mod}(A,B,a,b) \ \rhd\ \mathit{the}\ \ A\ \ \mathit{that}\ \ B[\emptyset/x].$$

7 Example: the axiom of parallels

To see how the sugaring principles work, take as an example the axiom of parallels in the formulation (written in lower level notation for readability)

$$(\Pi z : (\Sigma x : \mathrm{point})(\Sigma y : \mathrm{line})\mathrm{outside\_PL}(x,y))\mathrm{DAP}(p(z),p(q(z))),$$

where

$$\mathrm{outside\_PL}(a,b) \quad \triangleright \quad \textit{a lies outside b},$$

$$\mathrm{DAP}(a,b) = (\exists! x : \mathrm{line})(\mathrm{lie\_PL}(a,x) \& \mathrm{parallel}(x,b)) : \mathrm{prop} \text{ for } a : \mathrm{point}, b : \mathrm{line},$$

$$\mathrm{DAP}(a,b) \quad \triangleright \quad \textit{a determines a parallel to b}.$$

To find the different possibilities to express the axiom of parallels in English provided by our grammar, recall the definitional variants

every and if for Π,

Indef, some, and, that, and such_that for Σ,

A and there(A) for A : set,

Pron(A,a) and the(A,a) for $a : A$.

Start sugaring from the outermost form of the proposition. First choose the definitional variant every for Π. Then you must sugar the domain of quantification

$$(\Sigma x : \mathrm{point})(\Sigma y : \mathrm{line})\mathrm{outside\_PL}(x,y)$$

into a set expression. The only choice for the first Σ is a relative pronoun, that or such_that. The domain of this Σ must be sugared into the common noun *point*. The remaining part must be found a sentence-like expression. All ways of sugaring Σ are usable: if you choose the relative pronoun, just apply there. The domain of quantification of the axiom of parallels thus has the following sugarings, among others.

point that lies outside a line,

point that lies outside some line,

point such that there is a line and it lies outside it.

The third sugaring is a little strange, because the interpretation of the two occurrences of *it* seems not to be unique. The language of geometry overloads the verb *lie outside*, so that

$$\left.\begin{array}{l}\mathrm{outside\_PL}(x,y)\\ \mathrm{outside\_LP}(x,y)\end{array}\right\} \quad \triangleright \quad \textit{x lies outside y},$$

whence *it lies outside it* has two interpretations that, although equivalent, are distinct propositions. We cannot tell whether the sentence says that the point lies outside the line or that the line lies outside the point. But this explanation of the strangeness already contains the solution, which is to use definite noun phrases instead of pronouns,

point such that there is a line and the point lies outside the line.

To finish the first way of sugaring the axiom of parallels, we replace the first argument in

$$\mathrm{DAP}(p(z), p(q(z)))$$

by the quantifier phrase, as explained in Section 5, and the second argument by a pronoun or a definite phrase of the type line. Applying the sugaring rules for DAP and Pron and choosing the expression for the domain to be the first one cited above, gives

every point that lies outside a line determines a parallel to it as an unambiguous statement of the axiom of parallels. The word-to-word formalization of this sentence is the definitional variant

$$\text{every}(\text{that}(\text{point}, (x)\text{Indef}(\text{line}, (y)\text{outside\_PL}(x, y))), (z)\text{DAP}(p(z), \text{Pron}(\text{line}, p(q(z)))))$$

of the original proposition.

The reader can check that the proposition also has the following variants, and find some more of them.

if a point lies outside a line, it determines a parallel to it,

if there is a point such that there is a line such that the point lies outside the line the point determines a parallel to the line.

Observe that the two occurrences of *it* in the first variant are uniquely interpretable, because *determines a parallel to* is not overloaded. An early implementation of sugaring, written in Prolog by Petri Mäenpää, found 1128 variants of a donkey sentence with the same structure as the axiom of parallels.

8 Some uses of the plural

In the informal language of mathematics, it is often possible to find clear and unambiguous usages of linguistic structures that appear as hopelessly complex, if an unlimited fragment of natural language is taken under consideration. One such structure is the plural, which has been a persistent problem in logical semantics of Montague style. It has several uses that, when cooccurring, lead to multiple ambiguities. Mathematical texts still make unambiguous use of the plural, e.g. in the sentences

points A and B lie on the line a,

A and B are equal points,

all lines that pass through the center of a circle intersect its circumference.

The first of these sentences shows what von Plato (1993) defines as the term conjunction,

$$C(a.b) = C(a)\&C(b) : \text{prop for } A : \text{set},\ A : (A)\text{prop},\ a : A,\ b : A.$$

It is thus propositionally equal to the sentence

the point A lies on the line a and the point B lies on the line a,

in which no plural form occurs. In this case, the plural is just used for finding a more concise expression.

The second sentence does not employ the term conjunction, but it is propositionally equal to the singular sentence

A is equal to B

The difference between the first sentence and this one is an instance of the distinction between what is called distributive and nondistributive plural in linguistics. The distributive plural can be analyzed as a conjunction of singular instances, but the nondistributive plural cannot. For this particular sentence, we do have a nonplural equivalent, but I am not sure whether we always do.

The third sentence is propositionally equal to *every line that passes through the center of a circle intersects its circumference.*

Here there is no difference between *all lines* and *every line*, except the number agreement of the verb.

We have formulated a sugaring algorithm producing these uses of the plural (Ranta 1993, Chapter 9). In each of these cases, the plural forms of nouns and verbs are only produced in the sugaring process, and there is no type-theoretical operator corresponding to the plural. The rules we have discussed do not yet cover all uses of the plural in the informal language of mathematics. (But as long as we work in the direction of sugaring only, it makes no harm that all uses of an English mode of expression are not produced.) For instance, we do not yet quite understand the nondistributive use of the quantifier word *all* as in

all lines that pass through the center of a circle converge.

Nor do we quite understand the use of the plural pronoun *they*, which is sometimes distributive, paraphrasable by the term conjunction, e.g.

if A and B do not lie outside the line a, they are incident on it,

but sometimes used on the place of the "surface term conjunction", so that it fuses together the arguments of a predicate, e.g.

if a and b do not converge, they are parallel.

9 Problems and prospects

As indicated in the beginning of this paper, very little linguistic work has been done concerning the informal language of mathematics. To capture the essential structure of mathematical text, a grammatical representation of it should, at least, be able to express the mathematical propositions precisely. This can hardly be expected from all grammars in standard linguistics, but requires a grammatical formalism that comprises logic. Moreover, the formal and the informal language should be tied together by sugaring and parsing algorithms that satisfy the following condition.

> A correct informal proof results, when parsed, in a correct formal derivation, and vice versa.

There are two properties concerning ambiguity that can be stated. First,

> all expressions of the informal fragment are unambiguous.

But this is maybe too severe a condition. It makes little harm if the English fragment recognized contains ambiguities, if only the parser can detect them and ask the user to disambiguate. Instead, one can pose the weaker condition that

> every proposition of the formal theory can be expressed by an unambiguous English sentence.

A sugaring program satisfying this condition can provide a natural language interface to a formal proof system, stating theorems and their proofs in an easily readable form.

When considering mathematical language, instead of the fragment of everyday language familiar to the linguist, one soon realizes both a higher demand of unambiguity and a higher complexity of the propositions. There is still work to be done to find a sugaring algorithm that gives unambiguous expressions for all propositions of a formal theory. One particular problem is that the context in which a proposition is formed can be arbitrarily large, so that there are not enough anaphoric expressions to refer to each object uniquely. A very simple such context is created by the opening

given two lines, ...

formalizable by the quantifier

$$(\Pi z : (\Sigma x : line)line)$$

The anaphoric expressions that can be used for an arbitrary line are *it* and *the line*, but neither of these refers uniquely in this context. One way to solve this problem is to use the expressions *the first line*, *the second line*. Another one, much more idiomatic in mathematical language, is to introduce variables,

given two lines a and b, ...

whereafter reference can be made to *the line a* and to *the line b*. But this opening cannot be formalized as a quantifier, because the variable names are not usable outside the scope of the quantifier. The axiom of parallels in the formulation

if a point A lies outside a line a, A determines a parallel to a.

cannot thus be given the logical form we gave it in Section 7.

A more general defect of our Montague style grammar is that it only concerns propositions and not judgements, of which type theory has several forms that are all needed in precise formalization of mathematics. What we have done here only suffices for expressing axiomatic theories, in the format familiar from the metamathematical thinking of this century. Going beyond this format in mathematics, type theory also shows a model for grammar in general, to extend its views from propositions to judgements and other linguistic acts.

References

Noam Chomsky, *Syntactic Structures*, Mouton, The Hague, 1957.

Gottlob Frege, *Begriffsschrift*, Louis Nebert, Halle, 1879.

David Hilbert, *Grundlagen der Geometrie*, 1899; 2. Aufl. Teubner, Leipzig, 1903.

Per Martin-Löf, *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984.

Richard Montague, "English as a formal language", B. Visentini & al. (eds), *Linguaggi nella società e nella tecnica*, Edizioni di comunità, Milan, 1970. Reprinted in R. Montague, *Formal Philosophy*, Yale University Press, New Haven, 1974., .

Bengt Nordström, Kent Petersson and Jan Smith, *Programming in Martin-Löf's Type Theory. An Introduction*, Clarendon Press, Oxford, 1990.

Jan von Plato, "Axiomatization of geometry based on separation principles: point and line apartness and line convergence", paper manuscript, University of Helsinki, 1993.

Aarne Ranta, "Intuitionistic categorial grammar", *Linguistics and Philosophy* 14, pp. 203–239, 1991.

Aarne Ranta, “Type-Theoretical Grammar”, book manuscript, University of Helsinki, 1993. To appear in the Indices series at Oxford University Press.

Semantics for Abstract Clauses

D.A. Wolfram
*University of Oxford**

Abstract

We give declarative and operational semantics for logics that are expressible as finite sets of abstract clauses. The declarative semantics for these sets of generalized Horn clauses uses inductively defined sets and fixed points. It is shown to coincide with the operational semantics for successful and finitely failed derivations. The Abstract Clause Engine (ACE) implements proofs with abstract clauses. The semantics given here provide criteria for ACE's correctness and completeness.

1 Abstract Clauses

The approach here to the semantics of abstract clauses was presented first for two concrete examples: pure first-order logic programming [15], and the logic programming language based on the higher-order logic called The Clausal Theory of Types [17].

Experiments with the Abstract Clause Engine (ACE) [16], which implements abstract clauses, suggested that apart from logic programming, a generalization could be made which would also encompass simpler combinatorial problems and some relatively elaborate logical frameworks in a uniform way.

The semantics for this generalization relates the theorems of a logic expressed by abstract clauses to proof procedures for such a logic. It does not involve extra models such those of the first-order predicate calculus, Henkin-Andrews general models [2], or categorical models of the Calculus of Constructions [9]. Soundness and completeness results for logics which are based on such extra models can be combined with the results here, but we do not provide a general means to do so. Our concern is with whether a proof procedure for a logic expressed with abstract clauses can recursively derive just the theorems of the logic, and detect as non-theorems those formulas which can effectively be shown as such.

2 Terms and Substitutions

The meta-logic of abstract clauses is based on the monomorphic simply typed λ-calculus. This is not essential, but it seems to be sufficiently expressive for encoding calculi with dependent types and polymorphic type theories [12].

To define the meta-logic, we first provide some notations and definitions which are based on those of the simply typed λ-calculus [2, 3, 7]. We denote the sets of variables, and simply

*Author's address: Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD. E-mail: `David.Wolfram@prg`

typed λ-terms in β-normal form by $\mathcal{X}$, and $\mathcal{T}$, respectively, and the type of a term $t \in \mathcal{T}$ by $\tau(t)$. The set of free variables of a term $t \in \mathcal{T}$ is denoted by $\mathcal{F}(t)$.

Definition 2.1 A *substitution* is a function $\theta : \mathcal{X} \to \mathcal{T}$ which is written in postfix notation, and where $\forall x \in \mathcal{X} : \tau(x) = \tau(x\theta)$.

Notation 2.2 The domain of a substitution θ is written $D(\theta)$.

Notation 2.3 The symbols $\gamma, \mu, \pi, \rho, \sigma, \theta$, possibly with subscripts, denote substitutions. A substitution θ, for example, is represented by a set of the form $\{\langle x, x\theta\rangle \mid x \in D(\theta)\}$.

Definition 2.4 If for all $x \in D(\rho)$, $x\rho$ is a variable which is not x, then ρ is a *renaming substitution.*

Definition 2.5 Let V be any set of variables, and θ be any substitution. The *restriction* of θ to V is $\theta\lceil V = \{\langle x, x\theta\rangle \mid x \in D(\theta) \cap V\}$.

As a consequence of the Strong Normalization Theorem and the Church--Rosser Theorem, we can extend the definition of substitution to an endomorphism on $\mathcal{T}$.

Definition 2.6 The *instance* $t\theta$ of a term t by a substitution θ where

$$\theta\lceil\mathcal{F}(t) = \{\langle x_1, t_1\rangle, \ldots, \langle x_m, t_m\rangle\}$$

is the β-normal form of $(\lambda x_1 \cdots x_m.t)(t_1, \ldots, t_m)$.

A *closed instance* $t\theta$ of a term t is one where $\mathcal{F}(t\theta) = \emptyset$.

Definition 2.7 If σ and θ are substitutions then their *composition* $\sigma\theta$ is the substitution $\{\langle x, x\sigma\theta\rangle \mid x \in \mathcal{X}\}$.

3 Clausal Labelling Problems

We call the general form of problem encodable with abstract clauses a *clausal labelling problem.* Abstract clauses encode sequents:

$$\frac{u_1 \quad u_2 \quad \cdots \quad u_n}{l}$$

where the u_i are hypotheses and l is the conclusion. Such a sequent is represented by the abstract clause $l :- u_1, \ldots, u_n$. Similarly, the formula to be proved is a sequent of the form

$$\frac{u_1 \quad u_2 \quad \cdots \quad u_n}{}$$

which is represented by the clause $:- u_1, \ldots, u_n$.

Proofs are built up by composing sequents subject to a consistency test on the compositions. This partly involves pairing the conclusion of a sequent with an hypothesis of another sequent. A proof built from clauses is an finite collection of such pairs which collectively meet the consistency test. Building proofs can be seen as searching for conclusions of sequents, or labels, which enable such a collection to be formed.

We introduce some notations for such pairs before giving more formal definitions.

Definition 3.1 A *constraint* is a pair $\langle u, l\rangle$ where $u, l \in \mathcal{T}$. A *system* is a finite set of constraints.

Definition 3.2 A *positive clause* has the form $l :- u_1, \ldots, u_n$ where $l, u_i \in \mathcal{T}$, and $0 \leq i \leq n$. The term l is called the *head* of the clause, and the finite sequence $u_1, \ldots, u_n$ is called the *body* of the clause. The head and the terms u_i in the body of a positive clause are called the *terms* of the clause. The set of *free variables* of such a clause is the set $\mathcal{F}(l) \cup (\bigcup_{1\leq i\leq n} \mathcal{F}(u_i))$. When $n = 0$, the positive clause is written l.

A *negative clause* is a headless positive clause. It has the form $:- u_1, \ldots, u_n$. The terms u_i are called the *terms* of the negative clause. The set of *free variables* of such a clause is the set $\bigcup_{1\leq i\leq n} \mathcal{F}(u_i)$. When $n = 0$, the negative clause is written $\square$.

An *abstract clause* or *clause* is either a positive clause or a negative clause.

Using these definitions of clauses, we can now formally define clausal labelling problems and the consistency test.

Definition 3.3 A *clausal labelling problem* is a tuple $(\mathcal{U}, H, P, G_0)$ where

- $\mathcal{U} \subseteq \mathcal{T}$, and for every free variable x which occurs in a term in $\mathcal{U}$ where $\tau(x) \in T_0$, there is a constant symbol c which occurs in a term in $\mathcal{U}$ such that $\tau(x)$ is $\tau(c)$.

- H is a *test* which is a function whose domain is the set of all systems each of whose constraints is a pair of the form $\langle u, l\rangle$ where $u, l \in \mathcal{U}$, and whose range is the set of all countably infinite sets of substitutions each of which is an endomorphism on $\mathcal{U}$. It also satisfies the following conditions for every such system S.

  - H is *hereditary*: HS is non-empty if and only if for every subset $R \subseteq S$, HR is non-empty.
  - H is *sound*: for every $\mu \in HS$, $H\{\langle u\mu, l\mu\rangle \mid \langle u, l\rangle \in S\} = \{\emptyset\}$.
  - H is *complete*: for every substitution θ such that

  $$H\{\langle u\theta, l\theta\rangle \mid \langle u, l\rangle \in S\} = \{\emptyset\}$$

  there is $\mu \in HS$ and a substitution α such that $\theta = \mu\alpha$.

- P is a finite set of positive clauses. The terms of every clause in P are elements of $\mathcal{U}$.

- G_0 is a negative clause each of whose terms is an element of $\mathcal{U}$.

Notation 3.4 We shall sometimes abbreviate a positive clause of the form $l :- u_1, \ldots, u_n$ to $l :- B$ and possibly add subscripts, where B is syntactically identical to $u_1, \ldots, u_n$ and $n \geq 0$.

Many combinatorial problems are examples of clausal labelling problems.

Example 3.5 The distinct representatives problem is the problem of finding all tuples of choices of elements from finite sets so that in any tuple all of its elements are distinct.

As a clausal labelling problem, the terms in the goal are names of finite sets, P is the set of all pairings of a set name with an element from that set, and the test H ensures that the elements chosen from the set are distinct.

Example 3.6 Another such problem is the eight queens problem. The goal G_0 is $: -\ r_1, \ldots, r_8$ which represents the eight columns of a chessboard. P is the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$ which represents the rows of a chessboard. The test H ensures that no two queens placed on the board attack. For example $H\{\langle r_1, 1\rangle, \langle r_2, 3\rangle, \langle r_3, 5\rangle, \langle r_4, 2\rangle, \langle r_5, 4\rangle, \langle r_6, 6\rangle\} = \emptyset$ because the queens in the first and sixth columns attack each other.

However,

$$H\{\langle r_1, 1\rangle, \langle r_2, 7\rangle, \langle r_3, 5\rangle, \langle r_4, 8\rangle, \langle r_5, 2\rangle, \langle r_6, 4\rangle, \langle r_7, 6\rangle, \langle r_8, 3\rangle\} = \{\emptyset\}$$

because this is one of its ninety-two solutions

More generally, logic programming languages and some logical frameworks provide further examples.

Example 3.7 A pure first-order logic program P with a goal clause G_0 can be a clausal labelling problem provided that there is at least one 0-ary constant symbol in the terms of the program or goal.

The set $\mathcal{U}$ is the Herbrand Universe [5] for the program and goal, and the test H is first-order unifiability.

Example 3.8 An object logic with a theorem to be proved which is expressed using the meta-logic of Isabelle [11] is an example of a clausal labelling problem.

The program P is the set of rules of inference for the logic expressed as abstract clauses, and the goal G_0 represents the theorem to be proved. The signature of the logic must contain constant symbols whose types are the same as the types of the variables in the terms of P and G_0.

Higher-order unification corresponds to the test H, and higher-order unification procedures usually satisfy the soundness and completeness properties of Definition 3.3, (see Huet [7], for example).

4 Operational Semantics

We now define derivations for a clausal labelling problem. They involve renamed clauses.

Definition 4.1 A *renamed form* of a clause of the form $l :-\ u_1, \ldots, u_n$ is the clause $l\rho :-\ u_1\rho, \ldots, u_n\rho$ where ρ is a renaming substitution. Two clauses c_1 and c_2 are *renamed apart* if and only if $\mathcal{F}(c_1) \cap \mathcal{F}(c_2) = \emptyset$.

Definition 4.2 A *derivation* for $P \cup \{G_0\}$ of a clausal labelling problem $(\mathcal{U}, H, P, G_0)$ is a sequence $(G_0, W_0), (G_1, W_1), \ldots$ of pairs of goals and systems which is defined as follows.

Suppose that G_k is $:-\ u_1, \ldots, u_n$ where $k \geq 0$ and $n \geq 0$. If $k = 0$ then W_0 is $\emptyset$.

- If $n = 0$, the derivation is a *successful derivation of length* k, and the set of *answer substitutions* is $H(\bigcup_{0 \leq i \leq k} W_k)\}$.
- If $n > 0$, and there is an *input list* I_k of m_k clauses $l_j :-\ B_j$ where $1 \leq j \leq m_k \leq n$, which are

- any m_k clauses of P to which a renaming substitution has been applied so that they are renamed apart, and each is renamed apart from G_i and each clause in I_l where $0 \leq i \leq k$ and $1 \leq l < k$, and
- $H(\cup_{0 \leq i \leq k} W_i \cup \{\langle w_1, l_1\rangle, \ldots, \langle w_{m_k}, l_{m_k}\rangle\}) \neq \emptyset$ where $w_1, \ldots, w_{m_k}$ are m_k distinct terms in the terms of G_k which are called the *selected terms* of G_k

then, G_{k+1} is the goal clause formed by replacing w_j by B_j where $1 \leq j \leq m_k$, and W_{k+1} is the system $\{\langle w_1, l_1\rangle, \ldots, \langle w_{m_k}, l_{m_k}\rangle\}$

- Otherwise, the derivation is a *failed derivation of length k*.

The definition of derivation above has two special cases.

Definition 4.3 A derivation is a concurrent breadth-first or *BF-derivation* when m_k equals the number of terms of G_k where $k \geq 0$. It is depth-first or *DF-derivation* when $m_k = 1$. A *fair derivation* is either a failed derivation, or one in which every term of a goal is a selected term after a finite number of derivation steps.

A derivation tree represents a search space for a successful derivation.

Definition 4.4 The *derivation tree* for $P \cup \{G_0\}$ of a clausal labelling problem $(\mathcal{U}, H, P, G_0)$ is defined as follows.

1. The root of the derivation tree is $(G_0, \emptyset)$.
2. The children of (G_k, W_k) where $k \geq 0$ are all pairs of goal clauses and disagreement sets (G_{k+1}, W_{k+1}) which can be derived from (G_k, W_k) in one step.

Definition 4.5 A *successful branch* or *failed branch* of a derivation tree is a successful or failed derivation, respectively. If every branch of a derivation tree is a failed derivation, then the derivation tree is a *finitely failed derivation tree.*

A derivation tree is a *fair derivation tree* if every branch of it is a fair derivation. A derivation tree is a *finitely failed derivation tree* if every branch of it is a failed derivation.

A derivation tree is the *BF-tree* if every branch of it is a BF-derivation. A derivation tree is a *DF-tree* if every branch of it is a DF-derivation.

4.1 Derivation Tree Equivalences

We shall show equivalences of derivation trees defined in Definition 4.4.

Let T_1 and T_2 be derivation trees for a clausal labelling problem. The following algorithm traces a derivation $(G_0, W_0), (G_1, W_1), \ldots$ in T_2 from a given fair non-failed derivation in T_1. These derivations are called *equivalent* derivations.

Tracing Algorithm
Step $i \geq 0$.
Set I to the set of all immediate descendants of (G_i, W_i).
For every selected term u in G_i of T_2:

- If u appears in G_0 then trace the corresponding term down the given derivation in T_1 until it is a selected term which is replaced by a sequence B of terms and an equation $u = l$.

- Otherwise, u is introduced in some goal G_k of T_2 where $k \leq i$ in a body of a clause which replaces a term v. Find the corresponding term v in T_1 (this must have been done already) and trace down the given derivation in T_1 to where u is selected and replaced by a body B of a clause and an equation $u = l$.

- Delete from I all immediate descendants of (G_i, W_i) which do not have $l' :- B'$ in their input lists and $u = l' \in W_i$ where $l' :- B'$ is either a renamed form of $l :- B$, or $l :- B$. Call this new set I.

The single element in I is (G_{i+1}, W_{i+1}).

Definition 4.6 Two systems V and W are *equal up to a renaming of variables* if and only if there is a renaming substitution ρ such that the function $f : V \to W$ where $f\langle u, l\rangle = \langle u\rho, l\rho\rangle$ is a bijection.

Two substitutions θ_1 and θ_2 are *equal up to a renaming of variables* if and only if there is a renaming substitution ρ such that $\theta_1 = \theta_2\rho$.

Similarly, two sets Θ_1 and Θ_2 of answer substitutions are *equal up to a renaming of variables* if and only if there is a renaming substitution ρ such that the function $g : \Theta_1 \to \Theta_2$ where $g\theta = \theta\rho$ is a bijection.

Remark 4.7 For simplicity, from now on we shall say that two systems, two substitutions, or two sets of answer substitutions are equal if and only if they are equal up to a renaming of variables.

Tracing a term down the derivation in T_1 terminates because it is a fair derivation. If the derivation in T_1 is a successful derivation, then the traced derivation must also be finite. It is easy to verify that in this case, the union of all of the systems of the derivation in T_1, and the union of all of the systems of the traced derivation in T_2 are equal, and that the set of answer substitutions of both derivations are also equal. If the given derivation is an infinite derivation, then so is the traced derivation.

Lemma 4.8 *Let T_1 and T_2 be derivation trees for a clausal labelling problem.*

- *If T_1 has a successful branch then so does T_2.*

- *If T_1 has an infinite fair branch then T_2 has an infinite branch.*

These equivalences allow us to concentrate on the BF-tree.

Corollary 4.9 *The following statements are equivalent for derivations trees of a clausal labelling problem.*

- *The BF-tree has a successful derivation with a system which is equal to W.*

- *There is a derivation tree with a successful derivation with a system which is equal to W.*

- *Every derivation tree has a successful derivation with a system which is equal to W.*

The equivalence of fair derivation trees with respect to finite failure is shown by the following lemma which is a consequence of Lemma 4.8.

Lemma 4.10 *Let T_1 and T_2 be derivation trees for a clausal labelling problem. If T_1 is finitely failed and T_2 is a fair, then T_2 is finitely failed.*

Proof: Suppose that T_2 has a successful or (fair) infinite branch. As T_2 is fair, T_1 must have a successful or infinite branch by Lemma 4.8. This is a contradiction, and T_2 must be finitely failed. $\square$

Corollary 4.11 *The following statements are equivalent for a given clausal labelling problem.*

- *The BF-tree is finitely failed.*
- *There is a finitely failed derivation tree.*
- *Every fair derivation tree is finitely failed.*

5 Declarative Semantics

The declarative semantics of a clausal labelling problem are characterized by two inductively defined sets, and also by fixed points.

5.1 Inductive Definitions

We shall define the declarative semantics of a clausal labelling problem, and show that this semantics is equivalent to the operational semantics discussed above.

Definition 5.1 A clause is a *closed clause* if and only if its set of free variables is the empty set.

Let $(\mathcal{U}, H, P, G_0)$ be a clausal labelling problem. The set $|P|$ is the set of all closed clauses of the form $l\theta :- u_1\theta, \ldots, u_n\theta$ where $l :- u_1, \ldots, u_n$ is in P and θ is a substitution which is an endomorphism on $\mathcal{U}$.

Definition 5.2 Given a clausal labelling problem $(\mathcal{U}, H, P, G_0)$, its *base*, $b(P)$, is the set of all terms of the form $l\theta$ such that $l\theta \in |P|$ and there is a clause of the form $l :- B$ in P.

By the first part of Definition 3.3 of clausal labelling problem, such a base is never the empty set. We now define T_P.

Definition 5.3 Let $(\mathcal{U}, H, P, G_0)$ be a clausal labelling problem, and $D \subseteq |P|$. T_P is the function $2^{b(P)} \rightarrow 2^{b(P)}$ such that

$$\begin{aligned} T_P(D) \quad = \quad & \{a \mid (l :- u_1, \ldots, u_n \in |P|) \wedge (\mathcal{F}(a) = \emptyset) \wedge \\ & H\{\langle a, l\rangle, \langle u_1, d_1\rangle, \ldots, \langle u_n, d_n\rangle\} \neq \emptyset \wedge \\ & (d_i \in D) \text{ where } 1 \leq i \leq n\}. \end{aligned}$$

We now relate a clausal labelling problem to an inductive definition.

Definition 5.4 The *success set* is

$$S_P = \cup_{i \geq 0} {T_P}^i \emptyset$$

where ${T_P}^0\emptyset = \emptyset$, and ${T_P}^{k+1}\emptyset = T_P({T_P}^k\emptyset)$ and $0 \leq k < \omega$.

Kernels are the duals of such inductive definitions [1]. Here is the kernel of the preceding definition which defines another set of closed terms.

Definition 5.5 The *finite failure set* is

$$F_P = \cup_{i \geq 0} F_P^i \emptyset$$

where $F_P^0 \emptyset = \emptyset$, and $F_P^{k+1} \emptyset = b(P) - T_P(b(P) - (F_P^k \emptyset))$ and $0 \leq k < \omega$.

5.2 Fixed Points

A fixed point characterization of declarative semantics for clausal labelling problems uses classical results of Tarski [14] and Kleene [10] which are based on monotonic functions on complete lattices. It is defined here by using a set of closed positive clauses.

Proposition 5.6 *T_P is a monotonic function on $2^{b(P)}$: if $X \subseteq Y$, then $T_P(X) \subseteq T_P(Y)$ where $X, Y \in 2^{b(P)}$.*

Proposition 5.7 *$(2^{b(P)}, \subseteq)$ is a complete lattice, with $b(P)$ as top element, $\emptyset$ as bottom element, $\cup$ as join, and $\cap$ as meet.*

We shall use the following result [14].

Theorem 5.8 *(Tarski.) Let T be a monotonic function on elements of a complete lattice. Then T has a least fixed point* $\mathrm{lfp}(T)$*, and a greatest fixed point* $\mathrm{gfp}(T)$.

Corollary 5.9 *The function T_P on the complete lattice $(2^{b(P)}, \subseteq)$ has a least fixed point* $\mathrm{lfp}(T_P)$ *and a greatest fixed point* $\mathrm{gfp}(T_P)$.

Definition 5.10 Let L be a complete lattice and $T : L \to L$ be a mapping.

- $T \uparrow 0$ is the bottom element of L.
- $T \uparrow \omega$ is $\mathrm{lub}\{T \uparrow \beta \mid \beta < \omega\}$.
- $T \downarrow 0$ is the top element of L.
- $T \downarrow \omega$ is $\mathrm{glb}\{T \downarrow \beta \mid \beta < \omega\}$.

Definition 5.11 A function T on elements of a complete lattice L is a *continuous function* if $T(\mathrm{lub}(X)) = \mathrm{lub}(T(X))$ for every subset X of L all of whose finite subsets have an upper bound in X under the ordering $\subseteq$.

Proposition 5.12 *T_P is a continuous function on $(2^{b(P)}, \subseteq)$.*

The next theorem is the First Recursion Theorem [10]. It will link the inductive definition of the success set to the fixed point treatment.

Theorem 5.13 *(Kleene.)* $\mathrm{lfp}(T) = T \uparrow \omega$ *where T is a continuous function on elements of a complete lattice.*

As a result, we have:

Corollary 5.14 $\mathrm{lfp}(T_P) = T_P \uparrow \omega$.

The next definition links the finite failure set to the fixed point treatment.

Definition 5.15 $\overline{T_P \downarrow \omega} = b(P) - (T_P \downarrow \omega)$.

5.3 Summary

The following theorem states the equivalence between the inductively defined sets and fixed point constructions which were discussed above to define the declarative semantics of clausal labelling problems.

Theorem 5.16 *These are identities:*

- $S_P = \cup_{i\geq 0} S_P^i = \mathrm{lfp}(T_P) = T_P \uparrow \omega$.
- $F_P = \cup_{i\geq 0} F_P^i = \overline{T_P \downarrow \omega}$.

6 Soundness and Completeness

Definition 5.4 of success set, and Definition 5.5 of finite failure set correspond to successful and finitely failed BF-trees respectively. We shall prove the soundness and completeness of the BF-tree for successful and finitely failed derivations, and so the coincidence of operational and declarative semantics.

The following lemma is used in proving the completeness of the BF-tree for success, and its soundness for finite failure. Since the production of an element of $T_P^i\emptyset$, and $b(P) - F_P^{i+1}\emptyset$ where $i \geq 0$, is tantamount to a closed BF-derivation, by lifting such a derivation to the form of a BF-derivation, the results can be proved directly.

From now on, the symbols $\mathcal{U}$, H, P, and G_0 refer to an arbitrary but fixed clausal labelling problem $(\mathcal{U}, H, P, G_0)$. We also introduce the following notations.

Notation 6.1 We abbreviate

- $T_P^k\emptyset$ to S_k,
- $F_P^{k+1}\emptyset$ to F_k, and
- Y_k uniformly stands either for S_k, or for $b(P) - F_k$.

Lemma 6.2 *(Lifting Lemma.) Let $(G_0, W_0), \ldots, (G_j, W_j)$ be an initial sequence of a BF-derivation, G_j be the goal clause $:-\ u_1, \cdots, u_n$, and $\mu_i \in HW_{i+1}$ where $0 \leq i < j$. If there is a substitution α_j such that the terms of $G_j\mu_0 \cdots \mu_{j-1}\alpha_j$ are in Y_{k-j} where $k - j > 0$, then there is a BF-derivation step from (G_j, W_j) to (G_{j+1}, W_{j+1}) and a substitution α_{j+1} such that the terms of $G_{j+1}\mu_j\alpha_{j+1}$ are in Y_{k-j-1} and $\alpha_j\rho_j = \mu_j\alpha_{j+1}$.*

Proof: By definition of Y_{k-j}, there are n closed instances of clauses of P, $l_h\gamma_h :-\ B_h\gamma_h$ for $1 \leq h \leq n$, where $l_h :-\ B_h$ is a clause of P and the γ_h are closed substitutions such that $H\{\langle u_1\alpha_0, l_1\gamma_1\rangle, \ldots, \langle u_n\alpha_0, l_n\gamma_n\rangle\} \neq \emptyset$, and when $l > 0$ the terms of $B_h\gamma_h$ are in Y_{k-j-1}.

Let $\rho_j = \gamma_1 \ldots \gamma_n$ and $I = \{l_h :-\ B_h \mid 1 \leq h \leq n\}$. We can assume that the clauses in I are renamed apart, and each of them is renamed apart from all of the goal clauses $G_0, \ldots, G_j$.

This implies that $u_h\alpha_j\rho_j$ is $u_h\alpha_j$, and $l_h\alpha_j\rho_j$ is $l_h\gamma_h$ for every $h : 1 \leq h \leq n$. Therefore, $H\{\langle u_1\alpha_j\rho_j, l_1\alpha_j\rho_j\rangle, \ldots, \langle u_n\alpha_j\rho_j, l_n\alpha_j\rho_j\rangle\} \neq \emptyset$.

By the completeness of H from Definition 3.3 there is is $\mu_j \in HW_{j+1}$ and a substitution α_{j+1} such that $\alpha_j\rho_j = \mu_j\alpha_{j+1}$.

Hence, by Definition 4.3 of BF-derivation, there is a derivation step from (G_j, W_j) to (G_{j+1}, W_{j+1}) with input list $I_0 = I$, and when $k - j > 0$,

$$G_{j+1}\mu_0 \cdots \mu_{j-1}\alpha_j \text{ is } :-\ B_1\mu_0 \cdots \mu_{j-1}\alpha_j, \ldots, B_n\mu_0 \cdots \mu_{j-1}\alpha_j$$

and $\{B_1\mu_0 \cdots \mu_{j-1}\alpha_j, \ldots, B_n\mu_0 \cdots \mu_{j-1}\alpha_j\} \subseteq Y_{k-1}$. $\square$

6.1 Success Set

Suppose that the terms of a closed goal clause G_0 are in S_k. They are produced from elements in S_{k-1} which are in turn produced from elements in S_{k-2} and so on. From the definition of S_P, it is easy to construct a BF-derivation for $P \cup \{G_0\}$.

The following soundness theorem states that all terms of any closed instance of $G_0\mu$ belong to S_P where μ is an answer substitution for this derivation.

Theorem 6.3 *If $P \cup \{G_0\}$ has a successful BF-derivation of length k, then the terms of any closed instance $G_0\mu\alpha$ of $G_0\mu$ are in S_k, where μ is any answer substitution of the derivation.*

Proof: Apply $\mu\alpha$ to all the goal clauses of the derivation and apply any substitution so that all terms are closed. By Definition 4.3 of BF-derivation, and Definition 5.4 of S_P, if all terms of the instantiated goal clause G_n are in S_{i-1}, then all terms of G_{n-1} are in S_i.

As G_k is the empty goal clause, its terms are in $\emptyset = S_0$. □

The following theorem states the completeness of the BF-tree.

Theorem 6.4 *If there is a substitution α_0 such that the terms of $G_0\alpha_0$ are in S_k for a least $k > 0$, then $P \cup \{G_0\}$ has a successful BF-derivation of length k such that $G_0\alpha_0 = G_0\mu\gamma$, for an answer substitution μ and a substitution γ.*

Proof: Since the terms of $G_0\alpha_0 \in S_k$, by the leastness of k, after k repeated applications of the Lifting Lemma (Lemma 6.2), there are k BF-derivation steps from $(G_0, \emptyset)$ to (G_k, W_k) and substitutions α_j such that the terms of $G_j\mu_0 \cdots \mu_{j-1}\alpha_j$ are in S_{k-j} for $0 \leq j \leq k$.

From the Lifting Lemma, we have that $\alpha_i\rho_i = \mu_i\alpha_{i+1}$ where $1 \leq i < k$. Therefore, $\alpha_i\rho_i\rho_{i+1} = \mu_i\alpha_{i+1}\rho_{i+1}$, and $\mu_i\alpha_{i+1}\rho_{i+1} = \mu_i\mu_{i+1}\alpha_{i+2}$. It follows, using the associativity of the composition of substitutions [8], that $G_0\alpha_0 = G_0\alpha_0\rho_0 \cdots \rho_{k-1}$ and $G_0\alpha_0\rho_0 \cdots \rho_{k-1} = G_0\mu_0 \cdots \mu_{k-1}\alpha_k$. The substitution $\mu = \mu_0 \cdots \mu_{k-1}$ is an answer substitution, and $\gamma = \alpha_k$, as required. □

6.2 Finite Failure Set

The following theorem states the soundness of the BF-tree for finite failure.

Theorem 6.5 *If every BF-derivation for $P \cup \{G_0\}$ is failed by length $\leq k$, then every closed instance of G_0 contains a term in F_k.*

Proof: The proof is by induction on k. If every BF-derivation for $P \cup \{G_0\}$ is failed by length zero, then by definition of F_0 every closed instance of G_0 contains a term in F_0.

The induction hypothesis is that the theorem is true for all BF-derivations of length $k-1$. If every BF-derivation for G_0 is failed by length $\leq k$ then, by the induction hypothesis, every closed instance of every G_1 contains a term in F_{k-1}. Suppose there is a substitution α_0 such that the terms of $G_0\alpha_0$ are in $b(P) - F_k$. Then by the Lifting Lemma, there is a BF-derivation step from $(G_0, \emptyset)$ to (G_1, W_1) and a substitution α_1 such that the terms of $G_1\mu_0\alpha_1$ are in $b(P) - F_{k-1}$. This is a contradiction. Therefore every closed instance of G_0 contains a term in F_k. □

The next theorem states that the BF-tree is complete for finite failure.

Theorem 6.6 *If every closed instance of G_0 contains a term in F_k, then every BF-derivation for $P \cup \{G_0\}$ is failed by length $\leq k$.*

Proof: The proof is by induction on k. If every closed instance of G_0 contains a term in F_0, then every BF-derivation for $P \cup \{G_0\}$ is failed by length zero. Otherwise, by the definitions of BF-derivation and F_0, a closed instance of G_0 could be found which does not contain a term in F_0.

The induction hypothesis is that the theorem is true for all BF-derivations of length $k-1$. Let every closed instance of G_0 contain a term in $b(P) - F_k$. If there is a descendant (G_1, W_1) of $(G_0, \emptyset)$ and a substitution β such that the terms of $G_1\mu_0\beta$ are in $b(P) - F_{k-1}$, by the definitions of BF-derivation and F_k, the terms of a closed instance of $G_0\mu_0\beta$ are in $b(P) - F_k$ where $\mu_0 \in HW_1$.

This is a contradiction. Hence every closed instance of every descendant goal clause G_1 contains a term in F_{k-1} and by the induction hypothesis, every BF-derivation for $P \cup \{G_0\}$ is failed by length $\leq k$. □

6.3 Operational and Declarative Semantics

We now combine Theorem 5.16 for declarative semantics of clausal labelling problems with Theorems 6.3- 6.6 of the soundness and completeness of the BF-tree for successful and finitely failed derivations to show the coincidence of their operational and declarative semantics. This coincidence is extended to derivation trees in general by using Corollaries 4.9 and 4.11.

Theorem 6.7 *The following statements are equivalent.*

- *There is a substitution α such that the terms of $G_0\alpha$ are in S_P.*
- *There is a least $k > 0$ such that the terms of $G_0\alpha$ are in S_k.*
- *The BF-tree for $P \cup \{G_0\}$ has a successful branch of length k with answer substitution equal to μ, and there is a substitution β such that $G_0\alpha$ is $G_0\mu\beta$.*
- *There is a derivation tree for $P \cup \{G_0\}$ which has a successful branch with answer substitution equal to μ, and there is a substitution β such that $G_0\alpha$ is $G_0\mu\beta$.*
- *Every derivation tree for $P \cup \{G_0\}$ has a successful branch with answer substitution equal to μ, and there is a substitution β such that $G_0\alpha$ is $G_0\mu\beta$.*

Theorems 6.5 and 6.6 are now used in a similar way to characterize goals which do not have refutations.

Theorem 6.8 *The following statements are equivalent.*

- *Every closed instance of G_0 contains a term in F_P.*
- *There is a least $k > 0$ such that every closed instance of G_0 contains a term in F_k.*
- *The BF-tree for $P \cup \{G_0\}$ is finitely failed, and the length of none of its branches exceeds k.*
- *There is a finitely failed derivation tree for $P \cup \{G_0\}$.*
- *Every fair derivation tree for $P \cup \{G_0\}$ is finitely failed.*

7 Implementing Abstract Clauses

The Abstract Clause Engine (ACE) [16] implements proof searches for clausal labelling problems. It has been used for concrete problems such as combinatorial searches, first-order logic programming, and equational first-order logic programming with equational unification.

The tree that ACE searches is DF-tree. The user can specify how terms should be selected from goals. Theorems 6.7 and 6.8 provide implementation rules for ensuring that it is sound and complete with respect to the set of solutions to a clausal labelling problem, and with respect to the terms which can effectively be shown not to be solutions.

We expect that logical frameworks other than Isabelle [11] might be expressed as clausal labelling problems [4, 6, 13].

An advantage of working with ACE is that prototype proof procedures can be constructed relatively quickly. One just needs to implement the test, specify the syntax of the object logic, and sometimes to provide parsers and printers. ACE incorporates ten generic search methods and allows the depth of searches, number of solutions to be found, and number of nodes visited to be selected or recorded. These features should enable comparisons of search methods for an object logic to be made easily before a specific proof system is designed.

Acknowledgements

This work is supported by Christ Church, Oxford, and Oxford University Computing Laboratory.

References

[1] P. Aczel, An introduction to inductive definitions, in: *Handbook of Mathematical Logic*, (J. Barwise, Ed.), North-Holland, Amsterdam, 1977, 739–782.

[2] P.B. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic, Orlando, 1986.

[3] H.P. Barendregt, *The Lambda-Calculus: Its Syntax and Semantics*, Volume II, Second Edition, North-Holland, Amsterdam, 1984.

[4] Th. Coquand and G. Huet, The Calculus of Constructions, *Information and Computation* **76**, 2/3 (1988) 95–120.

[5] M.H. van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *Journal of the Association for Computing Machinery* **24** (1976) 733–742.

[6] R. Harper, F. Honsell, and G. Plotkin, A framework for defining logics, *Journal of the ACM* **40**, 1 (1983) 143–184.

[7] G.P. Huet, A unification algorithm for typed λ-calculus, *Theoretical Computer Science* **1** (1975) 27–57.

[8] G.P. Huet, *Résolution d'équations dans des Langages d'Ordre* $1, 2, \ldots, \omega$, Thèse de Doctorat d'Etat, Université Paris VII, Paris, 1976.

[9] J.M.E. Hyland and A.M. Pitts, The Theory of Constructions: categorical semantics and topos-theoretic models, *Contemporary Mathematics* **92** (1989) 137–199.

[10] S.C. Kleene, *Introduction to Metamathematics*, Van Nostrand, Princeton, 1952.

[11] L.C. Paulson, The foundation of a generic theorem prover, *Journal of Automated Reasoning* **5** (1989) 363–397.

[12] L.C. Paulson, Isabelle: the next 700 theorem provers, In *Logic and Computer Science*, (P. Oddifreddi, Ed.), Academic, Orlando, 1990, 361–386.

[13] F. Pfenning, Logic programming in the LF logical framework, in: *Logical Frameworks*, (G. Huet and G. Plotkin, Eds.), Cambridge, 1991, 149–181.

[14] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, *Pacific Journal of Mathematics* **5** (1955) 285–309.

[15] D.A. Wolfram, M.J. Maher, and J-L. Lassez, A unified treatment of resolution strategies for logic programs, *Proceedings of the Second International Logic Programming Conference*, Uppsala, 1984, 263–276.

[16] D.A. Wolfram, ACE: the abstract clause engine, System Summary, *Proceedings of the Tenth International Conference on Automated Deduction*, Kaiserslautern, Federal Republic of Germany, 23–27 July 1990, Lecture Notes in Artificial Intelligence **449**, Springer, Berlin, 1990, 679–680.

[17] D.A. Wolfram, *The Clausal Theory of Types*, Cambridge Tracts in Theoretical Computer Science **21**, Cambridge, 1993.