From semantics to rules: A machine assisted analysis

Catarina Coquand

Department of Computer Sciences.
Chalmers University of Technology and University of Göteborg.
S-412 96 Göteborg, Sweden.
e-mail catarina@cs.chalmers.se

1 Introduction

It is well known that it is easier to prove the correctness of a program if one derives the program from its specification rather than first writing the program and then prove the correctness. By analogy, we propose a method for proving completeness of an equational theory where the set of equations is found by interactively proving its completeness.

We will exemplify the method by two different theories. The first example is an extremely simple theory with only zero, successor and addition. Our main example is a simply typed λ-calculus with explicit substitution. We will for both examples show that the set of conversion rules we obtain by this method is sound and complete.

The method is to define an interpreter, $[\![M]\!]$, for the terms in the theory yielding the semantics of the terms in a model. Then we define an inversion function, *quote*, which returns a term corresponding to a given semantic object. This function should have the properties that for two terms having equal semantics it yields syntactically equal results. Now define a normalization function nf as.the composition of the interpreter and the inversion function. The name of this function may be a little bit misleading since it is not essential for the method that it actually returns a normal form, but this will be the case in the examples below. To obtain a complete set of conversion rules we set up the goal of proving that any term M converts with $nf(M)$, leaving the conversion relation yet undefined. Then we choose the conversion rules according to what we need to prove this theorem. Now this set of conversion rules is complete and we even have a decision procedure for it. Indeed if $[\![M]\!]$ and $[\![N]\!]$ are equal, then $quote([\![M]\!])$ and $quote([\![N]\!])$ are the same and hence M and N are convertible with each other.

The normalization function presented in this paper is similar to the one in [2]. In this paper they define a normalization function for simply typed λ-calculus with full $\beta\eta$ reduction to obtain a refinement of Friedman's completeness theorem. The main difference compared with their function and ours is that we extend this technique to λ-calculus with explicit contexts and substitutions and that we give the the complete set of conversion rules for this calculus. The treatment of variables is also different in our approach. The inversion function is also presented in [6] where it is discussed for combinatory logic. In [3] a more detailed study of how to extract the normalization function from a normalization proof using Tait's method is presented. One important difference with other works on explicit substitution is that we address the problem of using named variables.

2 The proof editor

The analysis is formalized in an implementation of Martin-Löf's type theory, ALF, which supports user-defined inductive definitions. This framework can be viewed as a functional programming language with strong typing rules. In this system theorems are identified with types and a proof is a function yielding a result in the type; all proofs are intuitionistic.

In Martin-Löf's framework we have the base type *Set* and the abstraction $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\alpha$ which is the type of n-ary functions $[x_1, \ldots, x_n]a$. The variable x_i will often be left out when it does not occur in $\alpha_{i+1}, \ldots, \alpha_n$ and α. For n-ary application we write $a(a_1, \ldots, a_n)$.

We will use natural deduction style when we present new inductive sets. The constructors of the set will be written in bold letters when they are introduced, but will in the rest of the text be written in ordinary style.

The terms we have in Martin-Löf's type theory are so called monomorphic, this means that all type information is in the term. For readability this information has been taken away in most of the terms.

We will use the intensional equality $Id(A, a, b)$ as described in [15] with its constructor $id(A, a) \in Id(A, a, a)$ (the A will be dropped) and the set *Bool* with its standard elements and operations. We also introduce a set, $T(b)$ depending on *Bool*, which has one element in $T(true)$ and no element in $T(false)$.

All the work presented below has been done in ALF and all the proofs of the theorems have been type checked. The proofs are using the pattern matching introduced by Coquand in [8] which is not a part of Martin-Löf's framework, but has been essential for carrying out this proof; in fact this study motivated the mechanism. ALF does not yet check that the theorems are proved using structural induction but it helps you to find all the cases in a case-analysis.

3 A small example

We present a very simple example to illustrate the method to obtain a complete set of conversion rules. We define a set of terms with the elements *zero*, *succ* and *add*. We define an evaluation function giving the semantics of these terms into the set of natural numbers in the obvious way. Now we write an inverting (injection) function from the set of natural numbers to the set of terms, again in the obvious way. The composition of the evaluation function and the injection is then a normalization function; we will call this function nf. One fundamental property of this function is that if M and N have the same semantics, then $Id(nf(N), nf(M))$

We now want to define the conversion rules between these terms in such a way that they are sound and complete with respect to the semantics, that is, two terms are convertible with each other if and only if their semantics are intensionally equal. To do this we set up the goal of proving that any term, M, is convertible with $nf(M)$, leaving the conversion rules yet undefined; in fact we could start off with the reflexivity, symmetry, transitivity and congruence rules since they will always be needed. We then choose our conversion rules according to what is needed to show this. We now have a complete set of equations. Indeed, if M and N have the same semantics, then $Id(nf(M), nf(N))$, and since M is convertible with $nf(M)$ and N is convertible with $nf(N)$ we get that M and N are convertible with each other.

Now we can use these results to show, firstly, that all terms are convertible with *zero* or a term of the form *succ* and secondly, that *succ* is one-to-one. Observe that we did not need the Church-Rosser property to get these results.

The main difference between this small example and the analysis of simply typed λ-calculus is that the model we choose has to be extensional for the λ-calculus. We will also get a somewhat stronger soundness result for the λ-calculus since we will have the soundness with respect to any model. Also, in order to prove that any term converts to the normal form we need to use the reducibility method for simply typed λ-calculus.

4 Simply typed λ-calculus with explicit substitutions

We are going to analyze a λ-calculus with explicit substitutions using named variables. We chose a non standard set of combinators for substitutions. We have an explicit projection function (that will be motivated later) and this time, contrary to the previous example, it is not at all clear a priori what a complete correct set of equations is for the calculus. We can really claim that the computer system helped us, not only in checking proofs, but also in the process of finding this set of rules.

There are several motivations why using a calculus with explicit substitutions:

- For reasons of efficiency and to avoid the problem of having to do α-conversion, implementations of the λ-calculus records the substitutions to be made rather than performing them immediately.
- For proving the correctness of these implementations it is not trivial to translate the implementation to the theoretical presentation. A λ-calculus with explicit substitutions has been suggested in [1, 12] to bring the theoretical presentation closer to the usual implementations.
- Explicit substitutions also correspond to let-expressions in functional language.

– There has been a recent interest in using them in proofs, eg. in ALF [11] this is implemented. The present proof makes use of explicit substitutions.

4.1 Definitions of the calculus

We define the set of simply typed derivation of λ-terms with named variables in the ordinary style à la Church, except that we have explicit substitution.

Definitions of types The types we have are base type and function types. The set of types $\mathcal{T} \in$ *Set* is introduced by:

$$\frac{}{\mathrm{o} \in \mathcal{T}} \qquad \frac{A, B \in \mathcal{T}}{\mathbf{fun}(A, B) \in \mathcal{T}}$$

Types will be denoted by A, B.

We recall that the constructors are written out in bold face only when they are introduced.

Definition of contexts To define context we will suppose an infinite set, *Name*, with names. We also suppose that we have a decidable equality on *Name*. A context is a list of names together with types and is defined mutually with a boolean-valued function *fresh* which describes when a name is fresh in a context:

$$\begin{array}{l} \mathcal{C} \in \mathit{Set} \\ \mathit{fresh} \in (\mathit{Name}; \mathcal{C})\mathit{Bool} \end{array}$$

The set of contexts $\mathcal{C}$ is introduced by:

$$\frac{}{\mathbf{empty} \in \mathcal{C}} \qquad \frac{\Gamma \in \mathcal{C} \qquad x \in \mathit{Name} \qquad A \in \mathcal{T} \qquad f \in T(\mathit{fresh}(x, \Gamma))}{\mathbf{add}(\Gamma, x, A, f) \in \mathcal{C}}$$

We will write [] for **empty** and $[\Gamma, x : A]$ for $\mathbf{add}(\Gamma, x, A, f)$, hence when we write $[\Gamma, x : A]$ it is implicit that x is fresh in Γ. The function *fresh* is defined by induction on the context as:

$$\begin{array}{ll} \mathit{fresh}(x, []) & \equiv \mathit{true} \\ \mathit{fresh}(x, [\Gamma, y : A]) & \equiv \mathit{and}(x \neq y, \mathit{fresh}(x, \Gamma)) \end{array}$$

We will use Γ, Δ and Θ for contexts.

The predicate *Occur* is true when a name with its type occurs in a context.

$$\frac{x \in \mathit{Name} \qquad A \in \mathcal{T} \qquad \Gamma \in \mathcal{C}}{\mathit{Occur}(x, A, \Gamma) \in \mathit{Set}}$$

The introduction rules are:

$$\frac{}{\mathbf{occ}_1 \in \mathit{Occur}(x, A, [\Gamma, x : A])} \qquad \frac{\mathit{occ} \in \mathit{Occur}(x, A, \Gamma)}{\mathbf{occ}_2(\mathit{occ}) \in \mathit{Occur}(x, A, [\Gamma, y : B])}$$

We also define the relation that describes when a context contains another.

$$\frac{\Gamma, \Delta \in \mathcal{C}}{Gt(\Gamma, \Delta) \in \ Set}$$

We use the notational convention $\Gamma \geq \Delta$ for $Gt(\Gamma, \Delta)$. The set $\geq$ has the constructors:

$$\frac{}{\mathbf{gt}_1 \in \Gamma \geq []} \qquad \frac{c \in \Gamma \geq \Delta \qquad occ \in Occur(x, A, \Gamma)}{\mathbf{gt}_2(c, occ) \in \Gamma \geq [\Delta, x : A]}$$

The following lemmas are easy to prove:

$$\begin{array}{l} Lemma1 \in (Occur(x, A, \Delta); \Gamma \geq \Delta)\ Occur(x, A, \Gamma) \\ Lemma2 \in (\Gamma \in \mathcal{C})\ \Gamma \geq \Gamma \\ Lemma3 \in (\Theta \geq \Gamma; \Gamma \geq \Delta)\ \Theta \geq \Delta \\ Lemma4 \in (\Gamma \in \mathcal{C}; x \in Name; A \in \mathcal{T})\ [\Gamma, x : A] \geq \Gamma \end{array}$$

Definition of derivations We will mutually inductively define the derivation of terms and substitutions

$$\frac{A \in \mathcal{T} \qquad \Gamma \in \mathcal{C}}{\mathcal{D}(\Gamma, A) \in \ Set} \qquad \frac{\Gamma, \Delta \in \mathcal{C}}{\mathcal{DS}(\Delta, \Gamma) \in \ Set}$$

We will use the notational convention $\Gamma \rightarrow A$ and $\Delta \rightarrow \Gamma$ for $\mathcal{D}(\Gamma, A)$ and $\mathcal{DS}(\Delta, \Gamma)$ respectively. A substitution for a context Δ over a context Γ is intuitively a list that associates to each name in Δ a derivation in Γ of the same type.

The derivations of terms are:

$$\frac{occ \in Occur(x, A, \Gamma)}{\mathbf{var}(occ) \in \Gamma \rightarrow A} \qquad \frac{\gamma \in \Delta \rightarrow \Gamma \qquad M \in \Gamma \rightarrow A}{\mathbf{subst}(M, \gamma) \in \Delta \rightarrow A}$$

$$\frac{M \in [\Gamma, x : A] \rightarrow B}{\mathbf{lambda}(M) \in \Gamma \rightarrow fun(A, B)} \qquad \frac{M \in \Gamma \rightarrow fun(A, B) \qquad N \in \Gamma \rightarrow A}{\mathbf{apply}(M, N) \in \Gamma \rightarrow B}$$

The explicit substitutions we have are the projection map, updating and composition:

$$\frac{c \in \Delta \geq \Gamma}{\mathbf{proj}(c) \in \Delta \rightarrow \Gamma} \qquad \frac{\gamma \in \Theta \rightarrow \Gamma \qquad \delta \in \Gamma \rightarrow \Delta}{\mathbf{comp}(\delta, \gamma) \in \Theta \rightarrow \Delta}$$

$$\frac{\gamma \in \Delta \rightarrow \Gamma \qquad M \in \Delta \rightarrow A}{\mathbf{update}(\gamma, M) \in \Delta \rightarrow [\Gamma, x : A]}$$

We will use the following notational conventions:

x_{Γ}^{A}	*for*	$\mathbf{var}(occ)$, *where* $occ \in Occur(x, A, \Gamma)$
$M\gamma$	*for*	$\mathbf{subst}(M, \gamma)$
$\lambda x.M$	*for*	$\mathbf{lambda}(M)$, *where* $M \in [\Gamma, x : A] \rightarrow B$
$M\ N$	*for*	$\mathbf{apply}(M, N)$
π_c	*for*	$\mathbf{proj}(c)$
$(\gamma, x = M)$	*for*	$\mathbf{update}(\gamma, M)$
$\delta\gamma$	*for*	$\mathbf{comp}(\delta, \gamma)$

Derivations of terms and substitutions will be named M, N and γ, δ, θ respectively.

The substitution π_c is not a standard primitive for explicit substitutions. Often one rather has an identity substitution (in $\Gamma \to \Gamma$) [1, 12] or the empty substitution (in $\Gamma \to []$) [6] in either case π_c would be derivable. Instead we have taken π_c as primitive and the identity and the empty substitution is then derivable. In [1] they also have a substitution $\uparrow$ which corresponds to a shift on substitutions; this substitution is here defined as π_c where $c \in [\Gamma, x : A] \geq \Gamma$. In the calculus [12, 19] we have as primitives also the thinning rules:

$$\frac{M \in \Gamma \to A \qquad c \in \Delta \geq \Gamma}{M \in \Delta \to A} \qquad \frac{\gamma \in \Delta \to \Gamma \qquad c \in \Theta \geq \Delta}{\gamma \in \Theta \to \Gamma} \qquad \frac{\gamma \in \Delta \to \Gamma \qquad c \in \Gamma \geq \Theta}{\gamma \in \Delta \to \Theta}$$

These rules will in our calculus be written as $M\pi_c$, $\gamma\pi_c$ and $\pi_c\gamma$ respectively. The first version of this work used these combinators, since we wanted this work to be a start for a complete mechanical analysis of this calculus. The set of conversion rules we obtained using these combinators suggested the use of π_c, which gives a more elegant set of conversion rules. There might be other advantages in using π_c: if a term is of the form $M\pi_c$ we know which the possible free variables are of the term M, information that might be used in a computation.

4.2 The semantics

Since we want to deal with full conversion on open terms and the η-rule, we choose to describe the semantics in a Kripke style model. See [7, 14, 10] for a discussion on Kripke models. A Kripke model is a set of possible worlds, $\mathcal{W}$, with a partial ordering, $\geq$, of extensions of worlds. We also have a family of ground sets $\mathcal{G}(w)$ over possible worlds which will be the interpretation of the base type.

Semantic objects We define the set of semantic objects as usual in Kripke semantics:

$$\frac{A \in \mathcal{T} \qquad w \in \mathcal{W}}{Force(w, A) \in Set}$$

$Force(w, A)$ will be written $w \Vdash A$. For the base type an element in $w \Vdash o$ is a family of elements in $\mathcal{G}(w')$, $w' \geq w$. For the type $fun(A, B)$ an element in $w \Vdash fun(A, B)$ is a family of functions from $w' \Vdash A$ to $w' \Vdash B$, $w' \geq w$.

The semantic objects will then be defined as

$$\frac{\begin{array}{c} f(c) \in \mathcal{G}(w') \\ [c \in w' \geq w\,] \end{array}}{\mathbf{g}(f) \in w \Vdash o} \qquad \frac{\begin{array}{c} f(c, u) \in w' \Vdash B \\ [c \in w' \geq w, u \in w' \Vdash A\,] \end{array}}{\mathbf{Lambda}(f) \in w \Vdash fun(A, B)}$$

We will use the notational convention $\Lambda(f)$ for $\mathbf{Lambda}(f)$.

We then define the following two elimination rules

$$appg \in (w \Vdash o; w' \geq w)\ \mathcal{G}(w')$$
$$app \in (w \Vdash fun(A, B); w' \geq w; w' \Vdash A)\ w' \Vdash B$$

in the obvious way. We will write $app_c(u, v)$ instead of $app(u, c, v)$, or even $app(u, v)$ in the case when $c \in w \geq w$.

The monotonicity function

$$mon \in (w \Vdash A; w' \geq w)\; w' \Vdash A$$

lifts a semantic object in one world into a semantic object in a bigger world and is defined by induction on the type. We will have the notational convention $mon_c(u)$ for $mon(u, c)$.

We now want to define an equality Eq on semantic objects. For the soundness of the η-rule we need $u \in w \Vdash fun(A, B)$ to be equal to $\Lambda([c, v]app_c(u, v))$, which corresponds to η-expansion on the semantical level. This means that the equality on our model must be extensional and that application and the monotonicity function commutes; i.e. lifting the result of an application up to a bigger world should be equal to first lifting the arguments and then do the application. We will say that a semantic object is uniform $\mathcal{U}$ if the application and monotonicity function commutes for this object. In [17] there is a further discussions on this commutativity. The predicates Eq and $\mathcal{U}$ will be mutually defined

$$\frac{u, v \in w \Vdash A}{Eq_{w,A}(u, v) \in Set} \qquad \frac{u \in w \Vdash A}{\mathcal{U}_{w,A}(u) \in Set}$$

They will both be defined by induction on the types, the idea of this way of defining extensionality is presented in [9]. Two semantic objects of base type are equal if they are intensionally equal and two semantic objects of function type are equal if the application of them to a uniform semantic object is extensionally equal. A semantic object of base type is always uniform. A semantic object of function type is uniform if it sends a uniform semantic object to a uniform semantic object, if it sends two equal uniform objects to equal semantic objects and if the application and monotonicity commutes for this semantic object.

The sets Eq and $\mathcal{U}$ is then (leaving out the constructors) defined by:

$$\frac{\begin{array}{l} Id(appg(u, c), appg(v, c)) \\ \qquad\qquad [c \in w' \geq w] \end{array}}{Eq_{w,o}(u, v)} \qquad \frac{\begin{array}{l} Eq_{w',B}(app_c(u_1, v), app_c(u_2, v)) \\ \qquad\qquad [c \in w' \geq w, \mathcal{U}_{w',A}(v)] \end{array}}{Eq_{w,fun(A,B)}(u_1, u_2)}$$

$$\frac{u \in w \Vdash o}{\mathcal{U}_{w,o}(u)} \qquad \frac{\begin{array}{l} \mathcal{U}_{w',B}(app_c(u, v))\ [c \in w' \geq w, \mathcal{U}_{w',A}(v)] \\ Eq_{w',B}(app_c(u, v_1), app_c(u, v_2)) \\ \quad [c \in w' \geq w, \mathcal{U}_{w',A}(v_1), \mathcal{U}_{w',A}(v_2), Eq_{w',A}(v_1, v_2)] \\ Eq_{w'',B}(mon_{c_1}(app_{c_2}(u, v)), app_{c_3}(u, mon_{c_1}(v))) \\ \quad [c_1 \in w'' \geq w', c_2 \in w' \geq w, c_3 \in w'' \geq w, \mathcal{U}_{w',A}(v)] \end{array}}{\mathcal{U}_{w,fun(A,B)}(u)}$$

The equality Eq is transitive and symmetric and for uniform objects it is also reflexive. Equal uniform values can be substituted in app and the function mon returns uniform objects for uniform input and equal results for equal input. We also need to prove the following properties about Eq and U:

- $(\mathcal{U}_{w,A}(u); c \in w \geq w)\ Eq_{w,A}(mon_c(u), u)$
- $(\mathcal{U}_{w,A}(u); c_1 \in w'' \geq w'; c_2 \in w' \geq w; c_3 \in w'' \geq w$
  $)\ Eq_{w'',A}(mon_{c_1}(mon_{c_2}(u)), mon_{c_3}(u))$
- $(\mathcal{U}_{w,fun(A,B)}(u); c \in w' \geq w; \mathcal{U}_{w',A}(v))\ Eq(app_c(u,v), app(mon_c(u), v))$

All the proofs are straightforward by induction on the type.

Semantic environments We define an environment

$$\frac{\Gamma \in \mathcal{C} \qquad w \in \mathcal{W}}{Force\_env(w, \Gamma) \in \ Set}$$

that associates a semantic object for each variable in a context. The set $Force\_env(w, \Gamma)$ will be written $w \mid\!\vdash \Gamma$. The set is introduced by:

$$\frac{}{\mathbf{empty\_env}(w) \in w \mid\!\vdash []} \qquad \frac{\rho \in w \mid\!\vdash \Gamma \qquad v \in w \mid\!\vdash A}{\mathbf{update\_env}(\rho, v) \in w \mid\!\vdash [\Gamma, x : A]}$$

We will write $\{\}_w$ instead of $\mathbf{empty\_env}(w)$ and $\{\rho, x = v\}$ instead of $\mathbf{update\_env}(\rho, v)$. We define the following operations on semantic environments.

$$\begin{aligned} &lookup \in (w \mid\!\vdash \Gamma; Occur(x, A, \Gamma))\ w \mid\!\vdash A \\ &mon \quad\ \in (w \mid\!\vdash \Gamma; w' \geq w)\ w' \mid\!\vdash \Gamma \\ &proj \quad\ \in (w \mid\!\vdash \Gamma, \Gamma \geq \Delta)\ w \mid\!\vdash \Delta \end{aligned}$$

Instead of $lookup(\rho, occ)$, $occ \in Occur(x, A, \Gamma)$, we write $lookup(\rho, x_\Gamma^A)$, instead of $mon(\rho, c)$ we write $mon_c(\rho)$ and instead of $proj(\rho, c)$ we write $proj_c(\rho)$.

We say that an environment is uniform $\mathcal{U}_{w,\Gamma}(\rho) \in \ Set\ [\rho \in w \mid\!\vdash \Gamma]$ if each semantic object in the environment is uniform. Two environments are equal $Eq_{w,\Gamma}(\rho_1, \rho_2) \in \ Set\ [\rho_1, \rho_2 \in w \mid\!\vdash \Gamma]$ if they are equal component-wise.

The equality on semantic environment Eq is transitive and symmetric and for uniform environments it is also reflexive. We can substitute equal semantic environments in $lookup$, mon, $proj$ and the result of applying these function to uniform environments is also uniform. We also need to prove the following properties about Eq and U for semantic environments:

- $(\mathcal{U}_{w,\Gamma}(\rho); c \in \Gamma \geq \Delta)\ Eq_{w,A}(lookup(\rho, x_\Gamma^A), lookup(proj_c(\rho), x_\Delta^A))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c \in w' \geq w)\ Eq_{w',A}(mon_c(lookup(\rho, x_\Gamma^A)), lookup(mon_c(\rho), x_\Gamma^A))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c_1 \in \Gamma \geq \Delta; c_2 \in [\Gamma, x : A] \geq \Delta$
  $)\ Eq_{w,\Delta}(proj_{c_2}((\rho, x = v)), proj_{c_1}(\rho))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c \in \Gamma \geq \Gamma)\ Eq_{w,\Gamma}(proj_c(\rho), \rho)$
- $(\mathcal{U}_{w,\Gamma}(\rho); c \in w \geq w)\ Eq_{w,\Gamma}(mon_c(\rho), \rho)$
- $(\mathcal{U}_{w,\Theta}(\rho); c_1 \in \Theta \geq \Delta; c_2 \in \Delta \geq \Gamma; c_3 \in \Theta \geq \Gamma$
  $)\ Eq_{w,\Gamma}(proj_{c_2}(proj_{c_1}(\rho)), proj_{c_3}(\rho))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c_1 \in w'' \geq w'; c_2 \in w' \geq w; c_3 \in w'' \geq w$
  $)\ Eq_{w'',\Gamma}(mon_{c_1}(mon_{c_2}(\rho)), mon_{c_3}(\rho))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c_1 \in \Delta \geq \Gamma; c_2 \in w' \geq w$
  $)\ Eq_{w',\Delta}(mon_{c_2}(proj_{c_1}(\rho)), proj_{c_1}(mon_{c_2}(\rho)))$

4.3 The semantics of the λ-calculus

We define evaluation functions for derivations of terms and substitutions in a given environment

$$\begin{aligned} [\![]\!]_{Term} &\in (\Gamma \to A; w \Vdash \Gamma)\ w \Vdash A \\ [\![]\!]_{Subst} &\in (\Delta \to \Gamma; w \Vdash \Delta)\ w \Vdash \Gamma \end{aligned}$$

The evaluation function for derivation corresponds via the Curry-Howard isomorphism to a proof of the soundness theorem of minimal implicational logic with respect to Kripke models. We also define an evaluation function for substitutions which computes a new environment.

We will only write $[\![]\!]$ below, it should be clear from the typing which one is used.

$$\begin{aligned} [\![x_\Gamma^A]\!]\rho &\equiv lookup(\rho, x_\Gamma^A) \\ [\![\lambda x.M]\!]\rho &\equiv \Lambda([c,u][\![M]\!]\{mon_c(\rho), x = u\}) \\ [\![MN]\!]\rho &\equiv app([\![M]\!]\rho, [\![N]\!]\rho) \\ [\![M\gamma]\!]\rho &\equiv [\![M]\!][\![\gamma]\!]\rho \\ [\![(\gamma, x = M)]\!]\rho &\equiv \{[\![\gamma]\!]\rho, x = [\![M]\!]\rho\} \\ [\![\gamma_1\gamma_2]\!]\rho &\equiv [\![\gamma_1]\!][\![\gamma_2]\!]\rho \\ [\![\pi_c]\!]\rho &\equiv proj_c(\rho) \end{aligned}$$

4.4 The inversion function

We now define an inversion function, *quote*, that given a semantic object in a particular Kripke model returns a term. The particular Kripke model that we choose is the one where as possible worlds we take contexts, as the order on worlds we take the order on contexts and as $\mathcal{G}$ we take the set of derivations of terms of base type. Via the Curry-Howard interpretation *quote* is a proof of the completeness theorem of minimal implicational logic with respect to this particular Kripke model, cf. the completeness proof in [14].

The function *quote* will be mutually defined with a function, *val*, which given a function from a context extension to a term gives a semantic object as result:

$$\begin{aligned} quote_{\Gamma,A} &\in (\Gamma \Vdash A)\ \Gamma \to A \\ val_{\Gamma,A} &\in (f \in (\Delta \in \mathcal{C}; \Delta \geq \Gamma)\ \Delta \to A)\ \Gamma \Vdash A \end{aligned}$$

Intuitively *quote* returns an η-expanded term and *val* returns an η-expanded semantic object.

We define an abbreviation for the semantic object corresponding to a variable.

$$var\_val_{\Gamma,A} \equiv [occ]val_{\Gamma,A}([\Delta, c]x_\Delta^A) \ \in (occ \in Occur(x, A, \Gamma))\ \Gamma \Vdash A$$

We write only $var\_val(x_\Gamma^A)$ for $var\_val_{\Gamma,A}(occ)$ where $occ \in Occur(x, A, \Gamma)$.

In order to do an η-expansion we need to suppose that from a given context we can choose a fresh name. So we will suppose a function $gensym \in (\Gamma \in \mathcal{C})Name$

and a proof $gf \in (\Gamma \in \mathcal{C})\ T(fresh(gensym(\Gamma), \Gamma))$ which proves that *gensym* returns a fresh variable.

The functions *quote* and *val* are both defined by induction on the type.

$$\begin{array}{ll}
quote_{\Gamma,o}(u) & \equiv appg(u, lemma2(\Gamma)) \\
val_{\Gamma,o}(f) & \equiv g(f) \\
quote_{\Gamma,fun(A,B)}(u) & \equiv \lambda z.quote_{[\Gamma,z:A],B}(app_c(u, var\_val(z^A_{[\Gamma,z:A]}))), \\
 & \quad \text{where } z \text{ is } gensym(\Gamma) \text{ and } c \text{ is } lemma4(\Gamma, z, A) \\
val_{\Gamma,fun(A,B)}(f) & \equiv \Lambda([c_1, v]val_B([\Theta, c_2]f(\Theta, c)\ quote_{\Theta,A}(mon_{c_2}(v)))) \\
 & \quad \text{where } c \text{ is } lemma3(c_2, c_1) \in \Theta \geq \Gamma
\end{array}$$

We also have that if two semantic objects in a Kripke model are extensionally equal, then the result of applying the inversion function to them are intensionally equal. To prove this we first have to show the following two lemmas:

- $(h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma; Id(f_1(\Delta, c), f_2(\Delta, c))))\ Eq(val_{\Gamma,A}(f_1), val_{\Gamma,A}(f_2))$
- $(c \in \Delta \geq \Gamma)\ Eq(mon_c(val_{\Gamma,A}(f)), val_{\Delta,A}([\Theta, c']f(\Theta, lemma3(c', c))))$

Both lemmas are proved by induction on the type.

The theorem

$$Theorem5 \in (Eq_{\Gamma,A}(u, v))\ Id(quote_{\Gamma,A}(u), quote_{\Gamma,A}(v))$$

is shown mutually with

$$Lemma6 \in (f \in (\Delta \in \mathcal{C}; \Delta \geq \Gamma)\ \Delta \rightarrow A)\ \mathcal{U}(val_{\Gamma,A}(f))$$

which states that *val* returns a uniform semantic object. Both the theorem and the lemma are proved by induction on the type.

We are now ready to define the function that computes a term to its normal form. For this we define the identity environment $id \in (\Delta \geq \Gamma)\ \Delta \Vdash \Gamma$ that for each variable in the context Γ associates the corresponding value of the variable in Δ (*var_val* gives the value of this variable). Now the normalization function, nf, is defined as the composition of the evaluation function and *quote*. This function is similar to the normalization algorithm in [2]; one difference is our use of Kripke models to deal with reduction under λ. One other difference is that, since we use explicit contexts we can use the context to find the fresh names in the *quote*-function.

To compute the normal form is now only to compute the semantics of the term in the identity environment and then invert the result:

$$nf(M)\ \equiv\ quote_{\Gamma,A}([\![M]\!]id(lemma2(\Gamma))))\ \in\ (\Gamma \rightarrow A)\ \Gamma \rightarrow A$$

Now we know by theorem theorem5 that nf returns the same result when applied to two derivations having the same semantics:

$$Corollary7 \in (Eq([\![M]\!]id, [\![N]\!]id))\ Id(nf(M), nf(N))$$

4.5 Completeness

We now want to define the complete set of conversion rules for derivations of terms and substitutions:

$$\frac{M, N \in \Gamma \to A}{Conv(M, N) \in \ Set} \qquad \frac{\gamma, \delta \in \Delta \to \Gamma}{Convs(\gamma, \delta) \in \ Set}$$

We use the notational convention $M \cong_{\Gamma,A} N$ and $\gamma \cong_{\Gamma,\Delta} \delta$ for $Conv(M, N)$ and $Convs(\gamma, \delta)$ respectively; or only $M \cong N$ and $\gamma \cong \delta$ when the typing is clear.

We now set up the goal of proving

$$Theorem8 \in (M \in \Gamma \to A)\ M \cong nf(M)$$

leaving the conversion rules yet undefined. We then choose our conversion rules according to what is needed to show this. Having done so we know that the set of conversion rules we obtain is complete since, by corollary7 we know that if $Eq([\![M]\!]id, [\![N]\!]id)$, then $Id(nf(M), nf(N))$ and then by theorem8 and symmetry and transitivity of $\cong$ we know that $M \cong N$. To prove the theorem we define a Kripke logical relation [14, 18]:

$$\frac{M \in \Gamma \to A \qquad u \in \Gamma \Vdash A}{\mathcal{CV}_{\Gamma,A}(M, u) \in \ Set}$$

this relation will correspond to Tait's computability predicate.

A derivation of base type is intuitively $\mathcal{CV}$-related to a semantic object of base type if they are convertible with each other; or more precisely:

$$\frac{M\pi_c \cong_{\Delta,A} appg(u, c)\ [c \in \Delta \geq \Gamma]}{\mathcal{CV}_{\Gamma,o}(M, u)}$$

A derivation of function type, $fun(A, B)$ is intuitively $\mathcal{CV}$-related to a semantic object of the same type if they send $\mathcal{CV}$-related derivations and objects of type A to $\mathcal{CV}$-related derivations and objects of type B; or more precisely:

$$\frac{\mathcal{CV}_{\Delta,B}((M\pi_c)N, app_c(u, v))\ [c \in \Delta \geq \Gamma, \mathcal{CV}_{\Delta,A}(N, v)]}{\mathcal{CV}_{\Gamma,fun(A,B)}(M, u)}$$

The idea of this predicate is that we can show that if $\mathcal{CV}(M, u)$, then $M \cong quote(u)$, hence if we show that $\mathcal{CV}(M, [\![M]\!]id)$, we have a proof of theorem8.

Correspondingly for the environment we define the set:

$$\frac{\gamma \in \Delta \to \Gamma \qquad \rho \in \Delta \Vdash \Gamma}{\mathcal{CV}\_env_{\Delta,\Gamma}(\gamma, \rho) \in \ Set}$$

We will write CV for $\mathcal{CV}\_env$.

The relation $\mathcal{CV}$ on substitutions has the introduction rules:

$$\frac{\gamma \in \Delta \to []}{\mathcal{CV}_{\Delta,[]}(\gamma, \{\}_\Delta)} \qquad \frac{\mathcal{CV}_{\Delta,\Gamma}(\pi_c\gamma, \rho) \qquad \mathcal{CV}_{\Delta,A}(x^A\gamma, u)}{\mathcal{CV}_{\Delta,[\Gamma,x:A]}(\gamma, \{\rho, x = u\})}$$

We prove that $\mathcal{CV}$ is closed under $\cong$, ie:

- $(M{\cong}N; \mathcal{CV}_{\Gamma,A}(N,u))\ \mathcal{CV}_{\Gamma,A}(M,u)$
- $(\gamma{\cong}\delta; \mathcal{CV}_{\Delta,\Gamma}(\delta,\rho))\ \mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho)$

We prove these lemmas by induction on A and Δ respectively. To illustrate the method we will show the case when A is o. We then have to show that $M\pi_c{\cong}appg(u,c)$ knowing that $N\pi_c{\cong}appg(u,c)$ and $M{\cong}N$. Since we define the conversion rules according to what we need to prove the lemma, we will introduce the rule:

$$\frac{M{\cong}_{\Gamma,A}N \qquad \gamma \in \Delta{\rightarrow}\Gamma}{M\gamma \cong_{\Delta,A} N\gamma}$$

and then by this rule and transitivity of conversion we get $M\pi_c{\cong}appg(u,c)$. We also have to prove the following lemmas:

- $(\mathcal{CV}_{\Gamma,A}(M,u); c \in \Delta \geq \Gamma)\ \mathcal{CV}_{\Delta,A}(M\pi_c, mon(u,c))$
- $(\mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho))\ \mathcal{CV}_{\Delta,A}(x_\Gamma^A\gamma, lookup(\rho, x_\Gamma^A))$
- $(\mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho); c \in \Theta \geq \Delta)\ \mathcal{CV}_{\Theta,\Gamma}(\gamma\pi_c, mon(\rho,c))$
- $(\mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho); c \in \Gamma \geq \Theta)\ \mathcal{CV}_{\Delta,\Theta}(\pi_c\gamma, proj(\rho,c))$

Now we are ready to prove that if γ and ρ are $\mathcal{CV}$-related, then $M\gamma$ and $[\![M]\!]\rho$ are $\mathcal{CV}$-related. This theorem corresponds to Tait's lemma saying that each term is computable under substitution. We prove this theorem mutually with a corresponding theorem for substitutions.

$$\begin{array}{ll} Theorem9 & \in (M \in \Gamma{\rightarrow}A; \mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho))\ \mathcal{CV}_{\Delta,A}(M\gamma, [\![M]\!]\rho) \\ Theorem10 & \in (\gamma \in \Delta{\rightarrow}\Gamma; \mathcal{CV}_{\Theta,\Delta}(\delta,\rho))\ \mathcal{CV}_{\Theta,\Gamma}(\gamma\delta, [\![\gamma]\!]\rho) \end{array}$$

Both theorems are proved by induction on the derivations.

The proof of that if M and u are $\mathcal{CV}$-related, then $M{\cong}quote(u)$ will be shown mutually with a corresponding lemma for val.

$$\begin{array}{ll} Theorem11 & \in (\mathcal{CV}(M,u))\ M{\cong}quote(u) \\ Lemma12 & \in (h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma)\ M\pi_c{\cong}f(\Delta,c))\ \mathcal{CV}_{\Gamma,A}(M, val_{\Gamma,A}(f)) \end{array}$$

In order to prove theorem8 we also prove that $\mathcal{CV}(\pi_c, id(c))$; then by this, theorem9 and theorem11 we get that $M\pi_c{\cong}_{\Gamma,A}nf(M)$, where $c \in \Gamma \geq \Gamma$. If we now define the conversion rule $M{\cong}_{\Gamma,A}M\pi_c$ for $c \in \Gamma{\rightarrow}\Gamma$ we get by transitivity of conversion that $M{\cong}nf(M)$.

Now it is easy to prove the completeness theorem

$$Theorem13 \in (Eq([\![M]\!]id, [\![N]\!]id))\ M{\cong}N$$

since by corollary7 we get that $Id(nf(m), nf(N))$ and by theorem8 and by transitivity and symmetry of conversion we get $M{\cong}N$.

The conversion rules for derivations of terms we have obtained by this method are the reflexivity, symmetry, transitivity, congruence rules and the following rules:

$$\begin{array}{lll}
(\lambda x.M)\gamma\ N & \cong M\ (\gamma, x = N) & M \in [\Gamma, x : A]{\rightarrow}B,\ \gamma \in \Delta{\rightarrow}\Gamma \\
M & \cong \lambda x.(M\pi_c\ x^A_{[\Gamma,x:A]}) & c \in [\Gamma, x : A] \geq \Gamma \\
x^A_{[\Gamma,x:A]}(\gamma, x = M) & \cong M & \gamma \in \Delta{\rightarrow}\Gamma, M \in \Delta{\rightarrow}A \\
x^A_\Gamma \pi_c & \cong x^A_\Delta & c \in \Delta \geq \Gamma \\
M\pi_c & \cong M & c \in \Gamma \geq \Gamma \\
(M\ N)\gamma & \cong (M\gamma)(\ N\gamma) & \\
(M\gamma)\delta & \cong M(\gamma\delta) &
\end{array}$$

The first rule is the β-rule and the second is the η-rule.

The conversion rules for derivations of substitutions we have obtained are the reflexivity, symmetry, transitivity, congruence rules and the following rules:

$$\begin{array}{lll}
(\gamma\delta)\theta & \cong \gamma(\delta\theta) & \\
(\gamma, x = M)\delta & \cong (\gamma\delta, x = M\delta) & \\
\pi_c(\gamma, x = M) & \cong \gamma & c \in [\Gamma, x : A] \geq \Gamma \\
\pi_{c_1}\pi_{c_2} & \cong \pi_{c_3} & c_2 \in \Theta \geq \Delta, c_1 \in \Delta \geq \Gamma, c_3 \in \Theta \geq \Gamma \\
\gamma\pi_c & \cong \gamma & c \in \Gamma \geq \Gamma
\end{array}$$

Completeness of the conversion rules for substitutions Actually the set of conversion rules that we have obtained for the explicit substitutions is not complete. For example, we have that π_{c_1} where $c_1 \in [x : A] \geq [x : A]$ and $(\pi_{c_2}, x = x^A)$ where $c_2 \in [x : A] \geq []$ have the same semantics but they are not convertible with each other. In order to find the complete set of conversion rules for the substitutions we define an inversion function for semantic environments.

$$quote_{\Delta,\Gamma} \in (\Delta \Vdash \Gamma)\ \Delta{\rightarrow}\Gamma$$

which is defined as:

$$\begin{array}{ll}
quote_{\Delta,[]}(\{\}_\Delta) & = \pi_{gt_1} \\
quote_{\Delta,[\Gamma,x:A]}(\{\rho, x = v\}) & = (quote_{\Delta,\Gamma}(\rho), x = quote_{\Delta,A}(v))
\end{array}$$

To define the normalization function for substitutions is now only to compute the semantics of the substitution in the identity environment and then invert the result

$$nf(\gamma)\ \equiv\ quote_{\Delta,\Gamma}([\![\gamma]\!]id(lemma2(\Gamma)))\ \in\ (\Delta{\rightarrow}\Gamma)\ \Delta{\rightarrow}\Gamma$$

If we now prove that all substitutions γ are convertible with $nf(\gamma)$, we can see that we also need the following two rules of conversion:

$$\begin{array}{ll}
\gamma \cong \pi_c & c \in \Delta \geq [],\ \gamma \in \Delta{\rightarrow}[] \\
\gamma \cong (\pi_c\gamma, x = x^A_{[\Gamma,x:A]}\gamma) & c \in [\Gamma, x : A] \geq \Gamma,\ \gamma \in \Delta{\rightarrow}[\Gamma, x : A]
\end{array}$$

These rules corresponds to the η-rule for terms in the sense that they both express that a derivation should be convertible with their canonical derivation.

4.6 Soundness of the conversion rules

In order to prove the soundness of the conversion rules we first have to show:

- $[\![M]\!]\rho$ and $[\![\gamma]\!]\rho$ are uniform if ρ is uniform
- $Eq([\![M]\!]\rho_1, [\![M]\!]\rho_2)$ and $Eq([\![\gamma]\!]\rho_1, [\![\gamma]\!]\rho_2)$ if $Eq(\rho_1, \rho_2)$
- $Eq(mon([\![M]\!]\rho, c), [\![M]\!]mon(\rho, c))$ and $Eq(mon_c([\![\gamma]\!]\rho), [\![\gamma]\!]mon_c(\rho))$

Then the proof of soundness

$$Theorem14 \in (M \cong_{\Gamma,A} N; \rho \in w \Vdash \Gamma)\ Eq([\![M]\!]\rho, [\![N]\!]\rho)$$

is straightforward by induction on the rules of conversion. Notice that this soundness result holds in any Kripke model.

4.7 Some consequences

We now have a decision algorithm for testing conversion between terms (together with a correctness proof).

We can now prove that the result of nf on a term of function type is a term in long η-normal form and for a term of base type it is a term in applicative normal form (a variable or an application where the first argument is on applicative normal form and the second argument on long η-normal form). Hence a term is convertible with its normal form.

It is also straightforward to show that λ is one-to-one up to α-conversion and this without using the Church-Rosser property.

5 Doing proofs on machine

One important aspect of this work is that this has been done on machine, actually this example was partly chosen because that it seemed to be possible to do it in a nice way in ALF. Often it is emphasized that a proof is machine checked, but this is only the minimum that you could ask of a proof system. Another important aspect is that the system helps you to develop your proof and I feel that ALF is on the right way for this. This work was not done by pen and paper first and then typed into the machine, but it has from the start been done in interaction with the system. Once getting used to the system and having introduced the definitions, I now find it more cumbersome to do the proofs by hand.

6 Acknowledgments

Thierry Coquand has given me the mathematical ideas of this proof and my work has been to analyze the complete formalization of these ideas and also to find a good formulation of simply typed λ-calculus which facilitates the mechanical proof.

Peter Dybjer and my supervisor Jan Smith have given me good comments on my work and on this paper.

References

1. M. Abadi, L. Cardelli, P-L. Curien and J-J Lévy. *Explicit Substitutions.* ACM Conference on Principles of Programming Languages, San Francisco, 1990.
2. U. Berger and H. Schwichtenberg. *An inverse of the evaluation functional for typed λ-calculus.* Proceedings of LICS 91.
3. U. Berger. *Program extraction from normalization proofs.* Proceedings of TLCA'93, LNCS vol. 664
4. V. Breazu-Tannen, Th. Coquand, C.A. Gunter and A. Scedrov. *Inheritance as implicit coercion.* Logic and Computation, feb 1991.
5. C. Coquand *Analysis of simply typed lambda-calculus in ALF.* Proceedings of the Winter Meeting, Tanum Strand, 1993.
6. Th. Coquand and P. Dybjer. *Intuitionistic Model Constructions and Normalization Proofs.* Paper in preparation.
7. Th. Coquand and J. Gallier. *A proof of strong normalization for the theory of constructions using a Kripke-like interpretation.* Proceedings of the first workshop in Logical Frameworks.
8. Th. Coquand. *Pattern Matching with Dependent Types.* Proceedings of the 1992 Workshop on Types for Proofs and Programs, eds B. Nordström, K. Petersson and G. Plotkin.
9. R. O. Gandy. *On the Axiom of Extensionality - Part I.* The Journal of symbolic logic, volume 21, March 1956, p 36 - 48.
10. S. A. Kripke *Semantical analysis of intutionistic logic 1* Formal systems and recursive functions, 1965, eds J.N. Crossley and M.A.E. Dummet, North-Holland.
11. L. Magnusson. *The new implementation of ALF.* Proceedings of the 1992 Workshop on Types for Proofs and Programs, eds B. Nordström, K. Petersson and G. Plotkin.
12. P. Martin-Löf. *Substitution calculus.* Handwritten notes, Göteborg, 1992.
13. P. Martin-Löf. *An Intuitionistic Theory of Types: Predicative Part.* Logic Colloquium '73, 1975,p. 73-118, eds. H. E. Rose and J. C. Shepherdson, North-Holland.
14. J. C. Mitchell and E. Moggi. *Kripke-style models for typed lambda calculus.* Annals for Pure and Applied Logic 51, 1991, p 99-124, North-Holland.
15. B. Nordström, K. Petersson and J. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.
16. Eike Ritter. *Categorical abstract machine for higher-order typed λ-calculus.* Phd thesis, Univ. of Cambridge, 30 Sept, 1992.
17. Dana S. Scott. *Relating theories of the lambda-calculus.* To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism, Academic Press, 1980.
18. R. Statman. *Logical Relation and the Typed λ-calculus.* Information and Control, 65, 1985.
19. A. Tasistro. Formulation of Martin-Löf's theory of types with explicit substitutions. Licentiate thesis, 1993.
20. R.D. Tennant. *Semantical analysis of specification logic.* Logics of programs. LNCS, vol. 193 Springer, 1985, p.373-386.