Higher-Order and Symbolic Computation, 15, 57–90, 2002

A Formalised Proof of the Soundness and Completeness of a Simply Typed Lambda-Calculus with Explicit Substitutions

CATARINA COQUAND catarina.coquand@cs.chalmers.se
Department of Computing Science, Chalmers University of Technology and University of Göteborg, 412 96 Göteborg, Sweden

Abstract. We present a simply-typed λ-calculus with explicit substitutions and we give a fully formalised proof of its soundness and completeness with respect to Kripke models. We further give conversion rules for the calculus and show also for them that they are sound and complete with respect to extensional equality in the Kripke model. A decision algorithm for conversion is given and proven correct. We use the technique "normalisation by evaluation" in order to prove these results. An important aspect of this work is that it is not a formalisation of an existing proof, instead the proof has been done in interaction with the proof system, ALF.

Keywords: formal methods, type theory, explicit substitutions, normalisation

1. Introduction

A precise way for giving semantics for a typed λ-calculus is by defining the semantics by induction on the derivation of the judgement $\Gamma \vdash t : A$ (meaning that term t is of type A in context Γ), as, for example, described in Mitchell's handbook chapter [14]. Semantics is thus given to terms-in-context—in practice, however, one might want to work with well-typed terms rather than terms-in-context. A semantics of well-typed terms can be based on that of terms-in-context, using an erasure function from terms-in-context to well-typed terms: the meaning of a well-typed term t can be defined as the meaning of those derivations whose erasure yields t. Such a definition is only meaningful if a coherence result can be shown: all derivations that erase to the same term must have the same semantics.

In this article, we present a fully formalized theory for a simply-typed λ-calculus with explicit substitutions, treating both terms-in-context and well-typed terms, and relating them. We have chosen to study a calculus with explicit substitutions for the following reasons:

– It is not trivial to relate implementations and traditional presentations of the λ-calculus. A λ-calculus with explicit substitutions has been suggested [1, 8, 13] to bring the theoretical presentation closer to the usual implementations.
– There has been a recent interest in using them in proof-systems. The language on which ALF [12] is based has explicit substitutions and the present proof makes use of them in the formalisation of the completeness theorem.
– Explicit substitutions with named variables can be used to give a precise solution to the problem of α-conversion.

An important aspect of this work is that it is a complete formalisation of proofs that were not known in advance. Instead, all proofs have been done in interaction with the proof editor, ALF [12].

1.1. Outline of the formalisation

In the following, we give a summary of the different parts of the formalisation.

- We formalize terms-in-context, $\Gamma \vdash t : A$, as a calculus of proof trees, $\Gamma \vdash A$, for implicational logic, i.e., term t is seen as a proof tree that derives A from the assumptions contained in Γ. A notion of equivalence of terms-in-context (including, e.g., β-reduction and η-conversion) is then defined with a conversion relation $\cong$ on these proof trees.
- We give a Kripke model with an ordered set of worlds and define when a proposition A is true for a world w in the model, $w \Vdash A$. We also define an extensional equality between objects in the model.
- We define an interpreter $[\![\ ]\!]$ that maps a proof tree in the theory to its semantics in the Kripke model. According to the Curry-Howard correspondence of programs to proofs and types to propositions, the type of the interpreter expresses soundness of the theory of proof trees with respect to the Kripke model. Consequently, the interpreter can be seen as a constructive soundness proof.
- We also define an inversion function, *reify*, which returns a proof tree corresponding to a given semantic object, yielding—again by the Curry-Howard isomorphism—a completeness proof.

  The inversion function is based on the principle of normalization by evaluation [3]. Obviously, for proof trees M and N with the same semantics, inversion yields the same proof tree $\mathit{reify}([\![M]\!]) = \mathit{reify}([\![N]\!])$. Consequently, we define a normalization function, nf, as the composition of the interpreter and the inversion function. We prove that if M is a proof tree, then $M \cong nf(M)$; together with the soundness result, this yields a decision algorithm for convertibility. We further show that nf returns a proof tree in long η-normal form.
- We prove that the convertibility relation on proof trees is sound and complete. The proof of soundness, i.e., convertible proof trees have the same semantics, is straightforward by induction on the structure of the trees. The completeness proof, i.e., if $[\![M]\!]$ and $[\![N]\!]$ are equal, then M and N are convertible, uses the fact that $\mathit{reify}([\![M]\!])$ is exactly the same as $\mathit{reify}([\![N]\!])$, and since $M \cong nf(M) \equiv \mathit{reify}([\![M]\!])$ and vice versa for N, we get that $M \cong N$.
- We define a calculus of simply typed λ-terms, a typed convertibility relation on the terms, and a deterministic reduction, $\Downarrow$. We also define an erasure function on proof trees that maps a proof tree M into a well-typed term M^-. For every well-typed term there is at least one proof tree that erases to this term; for defining the semantics of well-typed terms through that of proof trees, it suffices to show that all proof trees that erase to the same well-typed term have the same semantics.

  Because we know that convertible proof trees have the same semantics, it remains to show that all proof trees that erase to a given well-typed term are convertible. We use an argument due to Streicher [19]: we first prove that if $nf(M)^-$ and $nf(N)^-$ are the same, then $M \cong N$. Secondly, we prove that if a proof tree M erases to a well-typed term t,

then $t \Downarrow nf(M)^-$. Now, if two proof trees M and N erase to the same well-typed term t, then $t \Downarrow nf(M)^-$ and $t \Downarrow nf(N)^-$. Since the reduction is deterministic we have that $nf(M)^-$ and $nf(N)^-$ are the same, and hence $M \cong N$.
– We prove that the convertibility relation on proof trees is sound and complete, and give a decision algorithm for checking convertibility of two well-typed terms.

1.2. Related work

The normalisation function presented in our paper is similar to the one given by Berger and Schwichtenberg [3], who define a normalisation function for a simply typed λ-calculus with full $\beta\eta$-reduction in order to obtain a refinement of a completeness theorem due to Friedman [9]. 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 complete set of conversion rules for this calculus; another difference is that we use Kripke models. The inversion function is also presented by Coquand and Dybjer [5] where it is discussed for combinatory logic. Berger [2] carries out a more detailed study of how to extract the normalisation function from a normalisation proof using Tait's method.

2. The proof editor

The analysis is formalised in an implementation of Martin-Löf's type theory [12], ALF, which supports user-defined inductive definitions (see `http://www.cs.chalmers.se/Cs/Research/Logic/alf/guide.html` for more information about ALF). This framework can be viewed as a functional programming language with dependent types and user-defined data-types. It can not only be used for writing programs but also for expressing propositions and proofs. This is explained by the Curry-Howard isomorphism between sets and propositions.

The system consist of two parts; a proof engine and a graphical interface. The proofs are manipulated directly (through the graphical interface), unlike many other proof systems where proofs are done indirectly through tactics and where only the remaining subgoals are displayed. In the appendix some sample proofs are given both in the textual form (which is what the proof engine verifies) and a snapshot how this code would look like in the graphical interface.

2.1. Intuitionistic logic

The way we formulate logic in the framework is built on the Brouwer-Heyting-Kolmogorov explanation of the logical constants: a proposition is true in intuitionistic logic if and only if we have a method for proving it. Contrast this to classical logic where a proposition is supposed to be true or false. In intuitionistic logic we do not have that $A \vee \neg A$ holds in general since we do not have a method to prove it. Some examples of logical constants with their Brouwer-Heyting-Kolmogorov explanation and their interpretations as sets via the Curry-Howard isomorphism are:

– *Implication*. A proof of $A \supset B$ is a method (program) which to each proof of A yields a proof of B. The interpretation of $A \supset B$ in terms of sets is the set of functions from A to B, $A \to B$. The elements of the set $A \to B$ are of the form $\lambda x.b$, where $b \in B$ when $x \in A$.
– *Conjunction*. A proof of $A\&B$ is a proof of A and a proof of B. The interpretation of $A\&B$ is the Cartesian product of A and B, $A \times B$. The elements of the set is of the form $\langle a, b\rangle$ where $a \in A$ and $b \in B$.
– *Existential quantification*. A proof of $(\exists x \in A)B$ consists of an element $a \in A$ together with a proof of $B[a/x]$ (i.e., we have substituted a for x in B). The interpretation of $(\exists x \in A)B$ is the disjoint union of a family of sets, $(\Sigma x \in A)B$. The elements of the set are of the form $\langle a, b\rangle$ where $a \in A$ and $b \in B[a/x]$.

Example. If *fst* is the left projection on pairs, i.e., $fst\,(\langle a, b\rangle) = a$, then $\lambda x.fst\,(x)$ is a proof of $A\&B \supset A$.

2.2. *Martin-Löf's type theory*

Below we give a short introduction to Martin-Löf's type theory; a more thorough description can be found in the literature [7].

In Martin-Löf's framework we have the ground type *Set* (we explain below what the elements of *Set* are). Objects in type theory are formed from constants and variables using application and λ-abstraction. The λ-abstraction is written $[x_1, \ldots, x_n]a$: the typing rule for abstraction is:

$$[x_1, \ldots, x_n]a \in (x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\alpha,$$

if $a \in \alpha$ under the hypothesis that $x_1 \in \alpha_1, \ldots, x_n \in \alpha_n$. The variable x_i is often left out when it does not occur in $\alpha_{i+1}, \ldots, \alpha_n$ and α. The notation $x_1, x_2 \in \alpha$ means $x_1 \in \alpha; x_2 \in \alpha$. Application is written $a(a_1, \ldots, a_n)$ and we have that if $a \in (x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\alpha$, then $a(a_1, \ldots, a_n) \in \alpha\{a_1/x_1, \ldots, a_n/x_n\}$ (i.e., we substitute $a_1, \ldots, a_n$ for $x_1, \ldots, x_n$).

Example. Assume that $A, B \in Set$. The judgement $[x, y]x \in (A; B)A$ is derivable in Martin-Löf's type theory; logically, it can be interpreted as a proof of $A \supset B \supset A$, since if we have a proof of A (i.e., x) and if we have a proof of B (i.e., y), then we have a proof of A (which is x). Another way is to read this judgement is as the function $\lambda x.\lambda y.x$ of type $A \to B \to A$. In this judgement we assumed that A and B are two known types. We can generalize the judgement by abstracting over types A and B as $[A, B, x, y]x \in (A, B \in Set; A; B)A$ with the logical interpretation that for all propositions A and B we have that $A \supset B \supset A$.

We also have the possibility of introducing new constants in the theory. This we do by giving their definition; the form of these definitions is:

$$f \in A$$
$$f \equiv e$$

It is also possible to introduce a definition in a context $[x_1 \in A_1, \ldots, x_n \in A_n]$. If the context is empty, then it is left out. We can also instantiate the definition by an explicit substitution, $f\{x_1 := a_1, \ldots, x_n := a_n\}$. Explicit substitutions are used in the formalisation below. We shall however not get into the details of explicit substitutions in ALF here, but refer to the literature [12].

The objects of *Set* are inductively defined sets. They correspond to what in programming languages is called user-defined data-types. We define the elements in the sets by giving their constructors.

Example. The introduction rule for conjunction is written in natural deduction style as:

$$\frac{A \quad B}{A \& B}$$

When interpreted as a set we write the introduction rule as:

$$\frac{a \in A \quad b \in B}{\langle a, b \rangle \in A \times B}$$

In type theory we instead write:

$$And \in (A \in Set; B \in Set)Set$$

for $A \times B$ and

$$AndI \in (A \in Set; B \in Set, a \in A; b \in B)And(A, B)$$

for $\langle a, b \rangle$.

Since the elements in the set are inductively defined we can define functions by pattern matching on them (which corresponds both to proofs by induction and by case analysis).

Example. The natural-deduction–style rules for and-elimination are

$$\frac{A \& B}{A} \qquad \frac{A \& B}{B}$$

In Martin Löf's type theory we write these rules as:

$$fst \in (A, B \in Set; And(A, B))A$$
$$fst(A, B, AndI(A, B, a, b)) \equiv a$$
$$snd \in (A, B \in Set; And(A, B))B$$
$$snd(A, B, AndI(A, B, a, b)) \equiv b$$

There is no guarantee that the functions defined by pattern matching terminates. In the ALF system this has to be done by hand. It is however possible to write a test that guarantees termination, but such a test has not been implemented.

2.3. *Notes on the formalisation*

Much information has been taken away in the formal definitions below in order to increase their readability. When working in ALF one can also choose to hide this information in the graphical interface. Variables that occur freely in definitions and formulas are implicitly universally quantified.

We shall use natural deduction style when we present new inductive sets. When introducing constructors, we emphasize them by using bold letters.

Example. The formal definition of *And* reads

$$\frac{A \in Set \quad B \in Set}{And(A, B) \in Set}$$

with the introduction rule

$$\frac{A \in Set \quad B \in Set \quad a \in A \quad b \in B}{\mathbf{AndI}(A, B, a, b) \in And(A, B)}$$

or rather, hiding A and B for increased readability,

$$\frac{a \in A \quad b \in B}{\mathbf{AndI}(a, b) \in And(A, B)}$$

We use the intentional equality $\mathsf{Id}(A, a, b)$, where $a, b \in A$, as described by Nordström [16]. The introduction rule for this set is:

$$\frac{A \in Set \quad a \in A}{\mathbf{i}(A, a) \in \mathsf{Id}(A, a, a)}$$

The set A is left out in the rest of the formalisation.

We also use the sets $Bool$ and Σ, the existential quantifier, with their standard elements and operations. We further introduce a set, $T(b)$ depending on $Bool$, where $T(false)$ is absurdity and $T(true)$ is always true.

All the work presented below has been done in ALF and all the proofs of the theorems have been type checked. The proofs use the pattern matching introduced by Coquand [4], 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. This reflects the way we do proofs informally in operational semantics by induction and case analysis on derivations. ALF does not check that the theorems are proved using structural induction but it checks that all the cases in a proof by case-analysis are covered. Therefore the well-foundedness of the proofs below has been manually checked in order to guarantee total correctness.

3. The calculus of proof trees

We define the set of proof trees of implicational logic in the ordinary style à la Church, except that we use explicit substitutions.

3.1. *Definitions of types*

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

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

Types are denoted by A, B.

We recall that the constructors are written out in bold face only when they are introduced. We write $A \to B$ for $\mathit{fun}(A, B)$.

3.2. *Definition of contexts*

Suppose a countably infinite set, *Name*, with names together with a decidable equality on it. The set of contexts is mutually defined with a boolean-valued function *fresh* which describes when a name is fresh in a context:

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

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

$$\frac{}{\mathbf{empty} \in \mathcal{C}}$$

and

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

Ordinarily, the freshness condition is written as a side-condition, but since we are to formalise the proof trees, this information must be represented, too.

We 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 we also have a proof that x is fresh in Γ (when $[\Gamma, x : A]$ occurs in the conclusion of a statement, then it is implicit that $T(\mathit{fresh}(x, \Gamma))$ is an assumption.) The function *fresh* is defined by induction on the context as:

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

We use Γ, Δ and Θ for contexts.

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

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

The introduction rules are:

$$\frac{}{\mathbf{occ}_1 \in Occur(x, A, [\Gamma, x : A])}$$

and

$$\frac{occ \in Occur(x, A, \Gamma) \quad T(fresh(y, \Gamma))}{\mathbf{occ}_2(occ) \in Occur(x, A, [\Gamma, y : B])}$$

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:

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

The following lemmas are easy to prove:

$$\begin{aligned}
Lemma1 \ &\in \ (f \in (x \in Name; A \in \mathcal{T}; Occur(x, A, \Delta))Occur(x, A, \Gamma)) \\
&\qquad \Gamma \geq \Delta \\
Lemma2 \ &\in \ (Occur(x, A, \Delta); \Gamma \geq \Delta)\ Occur(x, A, \Gamma) \\
Lemma3 \ &\in \ (\Gamma \in \mathcal{C})\ \Gamma \geq \Gamma \\
Lemma4 \ &\in \ (\Theta \geq \Gamma; \Gamma \geq \Delta)\ \Theta \geq \Delta \\
Lemma5 \ &\in \ (\Gamma \in \mathcal{C}; x \in Name; A \in \mathcal{T}; T(fresh(x, \Gamma)))\ [\Gamma, x : A] \geq \Gamma \\
Lemma6 \ &\in \ (occ_1 \in Occur(x, A, \Gamma); occ_2 \in Occur(x, A, \Gamma) \\
&\qquad)\mathsf{Id}(occ_1, occ_2) \\
Lemma7 \ &\in \ (gt_1 \in \Gamma \geq \Delta; gt_2 \in \Gamma \geq \Delta)\mathsf{Id}(gt_1, gt_2)
\end{aligned}$$

Comment on Notation. As an example, to help understanding the formal notation above, we spell out *Lemma1*: given two contexts Γ, Δ, if for all names, x, and types, A, we have that $x : A$ occurs in Δ implies that $x : A$ occurs in Γ, then $\Gamma \geq \Delta$, i.e.,

$$\begin{aligned}
&\forall \Gamma, \Delta : \mathcal{C}.(\forall x : Name, A : \mathcal{T}.Occur(x, A, \Delta) \Rightarrow Occur(x, A, \Gamma)) \\
&\qquad \Rightarrow \Gamma \geq \Delta
\end{aligned}$$

Lemma1, *Lemma2* and *Lemma7* are proven by induction on Δ and *Lemma6* is proven by induction on Γ. *Lemma3* and *Lemma5* are direct consequences of *Lemma1* and for *Lemma4* we also use *Lemma2*. For the formal proofs of *Lemma1, Lemma2* and *Lemma3*, see the appendix.

The last two lemmas may seem slightly strange: they are used for guaranteeing independence of the proofs of *Occur* and $\geq$. For example, *Lemma6* says that if it can be shown that

$x : A$ occurs in a context Γ, then there is a unique proof of this fact. The need to prove independence of proofs might point to a problem in using type theory for formalising proofs. On the other hand, as we shall see, proof objects can also be useful: the present formalisation heavily uses the possibilities to perform case analysis on proof objects, which reduces the number of cases to consider.

3.3. *Definition of proof trees*

Proof trees and substitutions are mutually inductively defined by:

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

We use the notational convention $\Gamma \vdash A$ and $\Delta \to \Gamma$ for $\mathcal{D}(\Gamma, A)$ and $\mathcal{DS}(\Delta, \Gamma)$, respectively. A substitution of type $\Delta \to \Gamma$ intuitively is a list that associates to each $x : A$ in Γ a unique proof tree of type $\Delta \vdash A$.

The proof trees are defined by the following rules:

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

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

We recall that hidden assumptions in the definition above is implicitly universally defined and that the notation $[\Gamma, x : A]$ implies that x is fresh in Γ. For example; the full version of the definition:

$$\frac{M \in [\Gamma, x : A] \vdash B}{\mathbf{lambda}(M) \in \Gamma \vdash A \to B}$$

is

$$\frac{\begin{array}{l} A, B \in \mathcal{T} \\ x \in \mathit{Name} \\ \Gamma \in \mathcal{C} \\ T(\mathit{fresh}(x, \Gamma)) \\ M \in [\Gamma, x : A] \vdash B \end{array}}{\mathbf{lambda}(M) \in \Gamma \vdash A \to B}$$

In the definition of variables we can see that a proof of occurrence is part of the proof tree. The advantage is that we can do case-analysis on this proof to find out where in the context $x : A$ occurs. The disadvantage is that we need to prove that two variables are the

same even if they have two possibly different proofs of occurrence of $x : A$ (by *Lemma6* we know that the proofs are the same).

Explicit substitutions are built from a projection map, update and composition (see below for a discussion on the projection map):

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

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

We use the following notational conventions:

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

Proof trees and substitutions are 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, 13] or the empty substitution (in $\Gamma \to []$) [5]. Instead we have taken π_c as primitive. If $c \in \Gamma \geq \Gamma$, then π_c is the identity substitution and if $c \in \Gamma \geq []$, then π_c is the empty substitution. Abadi et al. [1] use a substitution $\uparrow$ that corresponds to a shift on substitutions; the same substitution is here defined as π_c where $c \in [\Gamma, x : A] \geq \Gamma$. In Martin-Löf's substitution calculus [13, 20] we have as primitives also thinning rules (i.e., if a term is well-typed in a given context, then it is also well-typed in a larger context and likewise for substitutions.) Here, thinning is achieved using π_c, since if, for example, $M \in \Gamma \vdash A$ and $c \in \Delta \geq \Gamma$, then $M\pi_c \in \Delta \vdash A$.

The first version of our work used combinators for the thinning rules, since we wanted it to be a start for a complete mechanical analysis of Martin-Löf's substitution calculus [13, 20]. The set of conversion rules we obtained using these combinators suggested the use of π_c, which gives fewer conversion rules. There might be other advantages in using π_c: if a proof tree is of the form $M\pi_c$ we know which are the possible free variables of the term M, information that might be used in a computation.

3.4. Convertibility of proof trees

The rules for conversion between proof trees and substitutions are inductively defined:

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

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

The conversion rules for proof trees are the reflexivity, symmetry, transitivity, congruence rules and the following rules:

$$
\begin{array}{ll}
(\lambda(x:A).M)\gamma\ N & \cong M\ (\gamma, x = N), \text{ if } M \in [\Gamma, x:A] \vdash B,\ \gamma \in \Delta \to \Gamma \\
M & \cong \lambda(x:A).\left(M\pi_c\ x^A_{[\Gamma, x:A]}\right), \text{ if } c \in [\Gamma, x:A] \geq \Gamma \\
x^A_{[\Gamma, x:A]}(\gamma, x = M) & \cong M, \text{ if } \gamma \in \Delta \to \Gamma,\ M \in \Delta \vdash A \\
x^A_\Gamma \pi_c & \cong x^A_\Delta, \text{ if } c \in \Delta \geq \Gamma \\
M\pi_c & \cong M, \text{ if } c \in \Gamma \geq \Gamma \\
(M\ N)\gamma & \cong (M\gamma)\ (N\gamma) \\
(M\gamma)\delta & \cong M(\gamma\delta) \\
(\gamma\delta)\theta & \cong \gamma(\delta\theta) \\
(\gamma, x = M)\delta & \cong (\gamma\delta, x = M\delta) \\
\pi_c(\gamma, x = M) & \cong \gamma, \text{ if } c \in [\Gamma, x:A] \geq \Gamma \\
\pi_{c_1}\pi_{c_2} & \cong \pi_{c_3}, \text{ if } c_2 \in \Theta \geq \Delta,\ c_1 \in \Delta \geq \Gamma,\ c_3 \in \Theta \geq \Gamma \\
\gamma\pi_c & \cong \gamma, \text{ if } c \in \Gamma \geq \Gamma \\
\gamma & \cong \pi_c, \text{ if } \gamma \in \Gamma \to [],\ c \in \Gamma \geq [] \\
\gamma & \cong \left(\pi_c\gamma, x = x^A_{[\Gamma, x:A]}\right), \\
 & \quad \text{if } \gamma \in \Delta \to [\Gamma, x:A],\ c \in [\Gamma, x:A] \geq \Gamma
\end{array}
$$

The first two rules correspond to the ordinary β- and η-rules, the next three define the effect of substitutions and the last two rules can be seen as the correspondence of the η-rule for substitutions. The remaining rules define how the substitutions distribute.

4. The semantic model

As 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 [6, 11, 15]. A Kripke model is a set of possible worlds, $\mathcal{W} \in Set$, with a partial ordering, $\geq\ \in (\mathcal{W}; \mathcal{W})Set$, of extensions of worlds. We also have a family of ground sets $\mathcal{G} \in (\mathcal{W})Set$ over possible worlds which are the interpretation of the base type. We also need independence of the proof of $\geq$, i.e., if $c_1, c_2 \in w' \geq w$, then $\mathsf{Id}(c_1, c_2)$. In ALF we define the model as the context in which the definitions and proofs are made.

4.1. *Semantic objects*

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

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

Force(w, A) is 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 $A \to B$ an element in $w \Vdash A \to B$ is a family of functions from $w' \Vdash A$ to $w' \Vdash B$, $w' \geq w$.

The semantic objects are defined by:

$$\frac{f \in (w' \in \mathcal{W}; w' \geq w)\mathcal{G}(w')}{\mathbf{Ground}(f) \in w \Vdash o}$$

and

$$\frac{f \in (w' \in \mathcal{W}; w' \geq w; w' \Vdash A)w' \Vdash B}{\mathbf{Lambda}(f) \in w \Vdash A \to B}$$

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

We define the following two elimination rules

$$\begin{array}{l} \mathit{ground} \in (w \Vdash o; w' \geq w)\mathcal{G}(w') \\ \mathit{ground}(\mathit{Ground}(f), c) \equiv f(w', c) \\ \mathit{app} \in (w \Vdash A \to B; w' \in \mathcal{W}; w' \geq w; w' \Vdash A)\ w' \Vdash B \\ \mathit{app}(\Lambda(f), w', c, u) \equiv f(w', c, u) \end{array}$$

We write $\mathit{app}_c(u, v)$ instead of $\mathit{app}(u, w', c, v)$, or even $\mathit{app}(u, v)$ in the case when $c \in w \geq w$.

The monotonicity function

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

lifts a semantic object in one world into a semantic object in a bigger world and is defined by induction on the type. We use the notational convention $\uparrow_c(u)$ for $\uparrow(u, c)$.

We also need to define an equality, Eq, on semantic objects. For the soundness of the η-rule we need $u \in w \Vdash A \to B$ to be equal to $\Lambda([w', c, v]\mathit{app}_c(u, v))$, 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 say that a semantic object is uniform $\mathcal{U}$ if the application and monotonicity function commutes for this object (see Scott [17] for a discussion regarding commutativity). The predicates Eq and $\mathcal{U}$ are mutually defined

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

They both are defined by induction on the types; this way of defining extensionality is presented by Gandy [10]. Two semantic objects of base type are equal if they are intentionally equal in all bigger worlds and two semantic objects of function type are equal if the application of them to a uniform semantic object in a bigger world 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 in a bigger world to a uniform semantic object, if it sends two extensionally equal uniform objects in a bigger world to extensionally equal semantic objects and if the application and monotonicity commutes for the semantic object.

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

$$\frac{(w' \in \mathcal{W}; c \in w' \geq w)\mathsf{Id}(ground(u, c), ground(v, c))}{Eq_{w,o}(u, v)}$$

$$\frac{(w' \in \mathcal{W}; c \in w' \geq w; \mathcal{U}_{w',A}(v))Eq_{w',B}(app_c(u_1, v), app_c(u_2, v))}{Eq_{w,A\to B}(u_1, u_2)}$$

$$\frac{u \in w \Vdash o}{\mathcal{U}_{w,o}(u)}$$

$$\frac{\begin{array}{l} h_0 \in (w' \in \mathcal{W}; c \in w' \geq w; \mathcal{U}_{w',A}(v))\mathcal{U}_{w',B}(app_c(u, v)) \\ h_1 \in (w' \in \mathcal{W}; c \in w' \geq w; \mathcal{U}_{w',A}(v_1); \mathcal{U}_{w',A}(v_2); Eq_{w',A}(v_1, v_2)) \\ \qquad Eq_{w',B}(app_c(u, v_1), app_c(u, v_2)) \\ h_2 \in (w' \in \mathcal{W}; w'' \in \mathcal{W}; c_1 \in w'' \geq w'; c_2 \in w' \geq w; c_3 \in w'' \geq w; \\ \qquad \mathcal{U}_{w',A}(v))Eq_{w'',B}(\uparrow_{c_1}(app_{c_2}(u, v)), app_{c_3}(u, \uparrow_{c_1}(v))) \end{array}}{\mathcal{U}_{w,A\to B}(u)}$$

The equality Eq is transitive and symmetric and it is reflexive for uniform objects. Equal uniform values can be substituted in *app* and the function $\uparrow$ returns uniform objects for uniform input and equal results for equal input. We also need to prove the following properties about Eq and U which are used in the proofs of soundness and completeness below:

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

All the proofs are straightforward by induction on the type.

4.2. Semantic environments

We define the set of environments:

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

where each variable in a context is associated with a semantic object. The set $Force\_env(w, \Gamma)$ is written $w \Vdash \Gamma$. The set is introduced by:

$$\frac{}{\mathbf{empty\_env}(w) \in w \Vdash []}$$

and

$$\frac{x \in \mathit{Name} \quad \rho \in w \Vdash \Gamma \quad v \in w \Vdash A}{\mathbf{update\_env}(\rho, v) \in w \Vdash [\Gamma, x : A]}$$

We 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:

$$\begin{aligned} \mathit{lookup} &\in (w \Vdash \Gamma; \mathit{Occur}(x, A, \Gamma))\ w \Vdash A \\ \mathit{lifte} &\in (w \Vdash \Gamma; w' \geq w)\ w' \Vdash \Gamma \\ \mathit{proje} &\in (w \Vdash \Gamma, \Gamma \geq \Delta)\ w \Vdash \Delta \end{aligned}$$

The function $\mathit{lookup}(\rho, \mathit{occ})$, $\mathit{occ} \in \mathit{Occur}(x, A, \Gamma)$, is defined by induction on the environment, ρ. We write $\mathit{lookup}(\rho, x^A_\Gamma)$ instead of $\mathit{lookup}\,(\rho, \mathit{occ})$. The function *lifte* that lifts an environment into a bigger world is also defined by induction on the environment ρ (we use the notation $\uparrow_c(\rho)$ instead of $\mathit{lifte}\,(\rho, c)$. The last function *proje* is the projection on environments and it is defined by induction on the proof of $\Gamma \geq \Delta$ (we write $\downarrow_c(\rho)$ instead of $\mathit{proje}\,(\rho, c)$).

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

The equality on semantic environments, Eq, is transitive, symmetric, and for uniform environments also reflexive. We can substitute equal semantic environments in *lookup*, $\uparrow_c$, $\downarrow_c$ and the result of applying these function to uniform environments is also uniform. We also need to prove the following properties about Eq for semantic environments which basically says that it doesn't matter in which order we lift and project the substitution:

- $(\mathcal{U}_{w,\Gamma}(\rho); c \in \Gamma \geq \Delta$
  $)\ Eq_{w,A}(\mathit{lookup}\,(\rho, x^A_\Gamma), \mathit{lookup}\,(\downarrow_c(\rho), x^A_\Delta))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c \in w' \geq w$
  $)\ Eq_{w',A}(\uparrow_c(\mathit{lookup}\,(\rho, x^A_\Gamma)), \mathit{lookup}\,(\uparrow_c(\rho), x^A_\Gamma))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c_1 \in \Gamma \geq \Delta; c_2 \in [\Gamma, x : A] \geq \Delta$
  $)\ Eq_{w,\Delta}(\downarrow_{c_2}(\{\rho, x = v\}), \downarrow_{c_1}(\rho))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c \in \Gamma \geq \Gamma)\ Eq_{w,\Gamma}(\downarrow_c(\rho), \rho)$
- $(\mathcal{U}_{w,\Gamma}(\rho); c \in w \geq w)\ Eq_{w,\Gamma}(\uparrow_c(\rho), \rho)$
- $(\mathcal{U}_{w,\Theta}(\rho); c_1 \in \Theta \geq \Delta; c_2 \in \Delta \geq \Gamma; c_3 \in \Theta \geq \Gamma$
  $)\ Eq_{w,\Gamma}(\downarrow_{c_2}(\downarrow_{c_1}(\rho)), \downarrow_{c_3}(\rho))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c_1 \in w'' \geq w'; c_2 \in w' \geq w; c_3 \in w'' \geq w$
  $)\ Eq_{w'',\Gamma}(\uparrow_{c_1}(\uparrow_{c_2}(\rho)), \uparrow_{c_3}(\rho))$
- $(\mathcal{U}_{w,\Gamma}(\rho); c_1 \in \Delta \geq \Gamma; c_2 \in w' \geq w)$
  $Eq_{w',\Delta}(\uparrow_{c_2}(\downarrow_{c_1}(\rho)), \downarrow_{c_1}(\uparrow_{c_2}(\rho)))$

These properties are used in the proofs of soundness and completeness below.

4.3. The semantics of the λ-calculus

We define evaluation functions for proof trees and substitutions in a given environment:

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

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

$$\begin{aligned}
[\![x^A_\Gamma]\!]\rho &\equiv lookup(\rho, x^A_\Gamma)\\
[\![\lambda(x:A').M]\!]\rho &\equiv \Lambda([w', c, u][\![M]\!]\{\uparrow_c(\rho), x = u\})\\
[\![MN]\!]\rho &\equiv app([\![M]\!]\rho, [\![N]\!]\rho)\\
[\![M\gamma]\!]\rho &\equiv [\![M]\!]([\![\gamma]\!]\rho)\\
[\![(\gamma, x = M)]\!]\rho &\equiv \{[\![\gamma]\!]\rho, x = [\![M]\!]\rho\}\\
[\![\gamma_1\gamma_2]\!]\rho &\equiv [\![\gamma_1]\!]([\![\gamma_2]\!]\rho)\\
[\![\pi_c]\!]\rho &\equiv \downarrow_c(\rho)
\end{aligned}$$

4.4. The inversion function

It is possible to go from the semantics back to the proof trees by an inversion function, *reify* that, given a semantic object in a particular Kripke model, returns a proof tree. The particular Kripke model that we choose has contexts as possible worlds, the order on contexts as the order on worlds, and $(\Delta \in \mathcal{C})\Delta \vdash o$ as $\mathcal{G}$. The instantiation is formalised in ALF by an explicit substitution.

In order to define the inversion function we need to be able to choose a unique fresh name given a context. We suppose a function $gensym \in (\Gamma \in \mathcal{C})Name$ and a proof of $(\Gamma \in \mathcal{C})$ $T(fresh(gensym(\Gamma), \Gamma))$ which proves that *gensym* returns a fresh variable. Note that *gensym* is a function taking a context as an argument and it hence always returns the same variable for a given context.

The function *reify* is quite intricate and we therefore start by oversimplifying the definition by forgetting the contexts and the order on them so that the intuition of how the algorithm works is clearer. The function *reify* is defined together with a function *val* that intuitively takes a proof tree of the form of an variable applied to zero or more arguments. The idea is that if $M \in \Gamma \vdash A_1 \to \ldots A_n \to o$, then

$$reify\,([\![M]\!]) \equiv \lambda(z_1 : A_1). \ldots \lambda(z_n : A_n).([\![M]\!]\ val(z_1) \ldots val(z_n))$$

where $[\![M]\!]\ val(z_1) \ldots val(z_n)$ is a proof tree of type o and $z_1, \ldots, z_n$ are fresh names, i.e.,

$$z_1 \equiv gensym(\Gamma), \ldots, z_n \equiv gensym([\Gamma, z_1 : A_1, \ldots, z_{n-1} : A_{n-1}]).$$

If x is a variable of type $A_1 \to \ldots A_k \to B$, then

$$val(x) = \Lambda([v_1] \ldots \Lambda([v_k](x\ reify\,(v_1) \ldots reify\,(v_n)))$$

(where Λ is the simplified interpretation of abstraction).

Example. Let $M \in [] \vdash (o \to o) \to o) \to (o \to o) \to o$ be

$$\lambda(x : ((o \to o) \to o).\lambda(y : o \to o).(x\ y)$$

then

$$\mathit{reify}\,([\![M]\!]) \equiv \lambda(z_1 : (o \to o) \to o).\lambda(z_2 : o \to o).([\![M]\!]\ \mathit{val}(z_1)\ \mathit{val}(z_2))$$

Observe that $[\![M]\!]\ \mathit{val}(z_1)\ \mathit{val}(z_2)$ is of type o and hence (again oversimplifying) a proof tree. We continue with

$$\begin{aligned}
&[\![M]\!]\ \mathit{val}(z_1)\ \mathit{val}(z_2) \equiv [\![x\ y]\!]\{x = \mathit{val}(z_1),\ y = \mathit{val}(z_2)\}\\
&\equiv \mathit{app}(\mathit{val}(z_1), \mathit{val}(z_2))\\
&\equiv \mathit{app}(\Lambda([v](z_1\ \mathit{reify}\,(v))), \mathit{val}(z_2)) \equiv z_1\ \mathit{reify}\,(\mathit{val}(z_2))
\end{aligned}$$

the proof tree $\mathit{reify}\,(\mathit{val}(z_2))$ is of type $o \to o$ so

$$\begin{aligned}
&\mathit{reify}\,(\mathit{val}(z_2)) \equiv \lambda(z_3 : o).\mathit{app}(\mathit{val}(z_2), \mathit{val}(z_3))\\
&\equiv \lambda(z_3 : o).\mathit{app}(\Lambda([v](z_2\ \mathit{reify}\,(v))), z_3)\\
&\equiv \lambda(z_3 : o).(z_2\ \mathit{reify}\,(z_3)) \equiv \lambda(z_3 : o).(z_2\ z_3)
\end{aligned}$$

Hence the result of $\mathit{reify}\,([\![M]\!])$ is

$$\lambda(z_1 : (o \to o) \to o).\lambda(z_2 : o \to o).(z_1\ \lambda(z_3 : o).(z_2\ z_3))$$

If two proof trees $M, N \in \Gamma \vdash A$ are equal up-to α-conversion, then the results of $\mathit{reify}\,([\![M]\!])$ and $\mathit{reify}\,([\![N]\!])$ are exactly equal since the same fresh variables are chosen by the *reify*-function.

The function *reify* is mutually defined with *val*, which given a function from a context extension to a proof tree returns a semantic object as result.

$$\begin{aligned}
\mathit{reify}_{\Gamma,A} &\in (\Gamma \Vdash A)\ \Gamma \vdash A\\
\mathit{val}_{\Gamma,A} &\in (f \in (\Delta \in \mathcal{C}; \Delta \geq \Gamma)\ \Delta \vdash A)\ \Gamma \Vdash A
\end{aligned}$$

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

$$\mathit{var\_val}_{\Gamma,A} \equiv [\mathit{occ}]\mathit{val}_{\Gamma,A}\left([\Delta, c]x^A_\Delta\right)\ \in (\mathit{occ} \in \mathit{Occur}(x, A, \Gamma))\ \Gamma \Vdash A$$

We write $\overline{x_{\Gamma,A}}$ for $\mathit{var\_val}_{\Gamma,A}(\mathit{occ})$ where $\mathit{occ} \in \mathit{Occur}(x, A, \Gamma)$.

The functions *reify* and *val* are both defined by induction on the type:

$$\begin{aligned}
\mathit{reify}_{\Gamma,o}(u) &\equiv \mathit{ground}(u, \mathit{Lemma3}(\Gamma))\\
\mathit{val}_{\Gamma,o}(f) &\equiv \mathit{Ground}(f)\\
\mathit{reify}_{\Gamma,A\to B}(u) &\equiv \lambda(z : A).\mathit{reify}_{[\Gamma,z:A],B}(\mathit{app}_c(u, \overline{z_{[\Gamma,z:A],A}})),\\
&\qquad \text{where } z \equiv \mathit{gensym}(\Gamma) \text{ and } c \equiv \mathit{Lemma5}(\Gamma, z, A)\\
\mathit{val}_{\Gamma,A\to B}(f) &\equiv \Lambda([\Delta, c_1, v]\mathit{val}_{\Delta,B}([\Theta, c_2]f(\Theta, c)\ \mathit{reify}_{\Theta,A}(\uparrow_{c_2}(v))))\\
&\qquad \text{where } c \equiv \mathit{Lemma4}(c_2, c_1) \in \Theta \geq \Gamma
\end{aligned}$$

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 intentionally equal. To prove this we first have to show the following two lemmas:

– $(h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma)\mathsf{Id}(f_1(\Delta, c), f_2(\Delta, c))$
  $)Eq(val_{\Gamma,A}(f_1), val_{\Gamma,A}(f_2))$
– $(c \in \Delta \geq \Gamma$
  $)Eq(\uparrow_c(val_{\Gamma,A}(f)), val_{\Delta,A}([\Theta, c']f(\Theta, Lemma4(c', c))))$

Both lemmas are proved by induction on the type and they are used in order to prove the theorem:

$$Theorem1 \in (Eq_{\Gamma,A}(u, v))\ \mathsf{Id}(reify_{\Gamma,A}(u), reify_{\Gamma,A}(v))$$

which is shown mutually with the lemma:

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

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 given a proof tree computes its normal form. For this we define the identity environment $id \in (\Delta \geq \Gamma)\ \Delta \Vdash \Gamma$ which to each variable in the context Γ associates the corresponding value of the variable in Δ (*var_val* gives the value of this variable). The normalisation function, *nf*, is defined as the composition of the evaluation function and *reify*. This function is similar to the normalisation algorithm given by Berger [3]; 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 *reify*-function.

The computation of the normal form is done by computing the semantics of the proof tree in the identity environment and then inverting the result:

$$nf(M) \equiv reify_{\Gamma,A}([\![M]\!]id(Lemma3(\Gamma)))) \in (\Gamma \vdash A)\ \Gamma \vdash A$$

We know by *Theorem 1* that *nf* returns the same result when applied to two proof trees having the same semantics:

$$Corollary1 \in (Eq([\![M]\!]id, [\![N]\!]id))\ \mathsf{Id}(nf(M), nf(N))$$

4.5. Soundness and completeness of proof trees

We have in fact already shown soundness and completeness of the calculus of proof trees.

The evaluation function, $[\![\ ]\!]$, for proof trees corresponds via the Curry-Howard isomorphism to a proof of the soundness theorem of minimal logic with respect to Kripke models. The function is defined by pattern matching which corresponds to a proof by case analysis of the proof trees.

The inversion function *reify* is, again via the Curry-Howard isomorphism, a proof of the completeness theorem of minimal logic with respect to a particular Kripke model where the worlds are contexts.

4.6. Completeness of the conversion rules for proof trees

In order to prove that the set of conversion rules is complete, i.e., $Eq([\![M]\!]id, [\![N]\!]id)$ implies $M \cong N$, we must first prove that:

$$Theorem2 \in (M \in \Gamma \vdash A)\ M \cong nf(M)$$

To prove the theorem we define a Kripke logical relation [15, 18]:

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

which corresponds to Tait's computability predicate.

A proof tree 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:

$$\frac{h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma) M\pi_c \cong_{\Delta,o} ground(u, c)}{\mathcal{CV}_{\Gamma,o}(M, u)}$$

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

$$\frac{h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma; \mathcal{CV}_{\Delta,A}(N, v))\mathcal{CV}_{\Delta,B}((M\pi_c)\ N, app_c(u, v))}{\mathcal{CV}_{\Gamma,A\to B}(M, u)}$$

The idea of this predicate is that we can show that if $\mathcal{CV}(M, u)$, then $M \cong reify(u)$, hence if we show that $\mathcal{CV}(M, [\![M]\!]id)$, we have a proof of *Theorem2*.

Correspondingly for the environment we define the set:

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

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

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

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

In order to prove *Lemma 8* below we need to prove the following properties about $\mathcal{CV}$:

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

– $(\mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho))\ \mathcal{CV}_{\Delta,A}(x^A_\Gamma\gamma, \mathit{lookup}\,(\rho, x^A_\Gamma))$
– $(\mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho); c \in \Theta \geq \Delta)\ \mathcal{CV}_{\Theta,\Gamma}(\gamma\pi_c, \uparrow_c(\rho))$
– $(\mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho); c \in \Gamma \geq \Theta)\ \mathcal{CV}_{\Delta,\Theta}(\pi_c\gamma, \downarrow_c(\rho))$

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 lemma corresponds to Tait's lemma saying that each term is computable under substitution. We prove the lemma:

$$\mathit{Lemma8} \in (M \in \Gamma \vdash A; \mathcal{CV}_{\Delta,\Gamma}(\gamma,\rho))\mathcal{CV}_{\Delta,A}(M\gamma, [\![M]\!]\rho)$$

mutually with a corresponding lemma for substitutions

$$(\gamma \in \Delta\rightarrow\Gamma; \mathcal{CV}_{\Theta,\Delta}(\delta,\rho))\ \mathcal{CV}_{\Theta,\Gamma}(\gamma\delta, [\![\gamma]\!]\rho)$$

Both lemmas are proved by induction on the proof trees using the lemmas above.

The lemma:

$$\mathit{Lemma9} \in (\mathcal{CV}(M,u))\ M \cong \mathit{reify}\,(u)$$

is shown mutually with a corresponding lemma for *val*:

$$(f \in (\Delta \in C; c \in \Delta \geq \Gamma)\ \Delta \vdash A$$
$$; h \in (\Delta \in C; c \in \Delta \geq \Gamma)\ M\pi_c \cong f(\Delta, c))\ \mathcal{CV}_{\Gamma,A}(M, \mathit{val}_{\Gamma,A}(f))$$

In order to prove *Theorem2* we also prove that $\mathcal{CV}(\pi_c, \mathit{id}(c))$; then by this, *Lemma8* and *Lemma9* we get that $M\pi_c \cong_{\Gamma,A} \mathit{nf}(M)$, where $c \in \Gamma \geq \Gamma$. Using the conversion rule $M \cong_{\Gamma,A} M\pi_c$ for $c \in \Gamma \geq \Gamma$ we get by transitivity of conversion of $\cong$ that $M \cong \mathit{nf}(M)$.

It is now easy to prove the completeness theorem

$$\mathit{Theorem3} \in (\mathit{Eq}([\![M]\!]\mathit{id}, [\![N]\!]\mathit{id}))\ M \cong N$$

Proof: We know by *Corollary1* that $\mathsf{Id}(\mathit{nf}(M), \mathit{nf}(N))$ and then by *Theorem2* and symmetry and transitivity of $\cong$ we get that $M \cong N$. □

4.7. Completeness of the conversion rules for substitutions

The proof of completeness above does not imply that the set of conversion rules for substitutions is complete. In order to prove the completeness of conversion rules for the substitutions we define an inversion function for semantic environments:

$$\mathit{reify}_{\Delta,\Gamma} \in (\Delta \Vdash \Gamma)\ \Delta\rightarrow\Gamma$$

which is defined as:

$$\mathit{reify}_{\Delta,[]}(\{\}_\Delta) = \pi_{gt_1}$$
$$\mathit{reify}_{\Delta,[\Gamma,x:A]}(\{\rho, x = v\}) = (\mathit{reify}_{\Delta,\Gamma}(\rho), x = \mathit{reify}_{\Delta,A}(v))$$

The normalisation function on substitutions is defined as the inversion of the semantics of the substitution in the identity environment

$$nf(\gamma) \equiv reify_{\Delta,\Gamma}([\![\gamma]\!]id(Lemma3(\Gamma))) \in (\Delta \to \Gamma)\ \Delta \to \Gamma$$

The completeness result for substitutions follows in the same way as for proof trees, i.e., we must prove that $\gamma \cong nf(\gamma)$.

4.8. 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 $Eq(\rho_1, \rho_2)$
- $Eq(\uparrow_c([\![M]\!]\rho), [\![M]\!]\uparrow_c(\rho))$ and $Eq(\uparrow_c([\![\gamma]\!]\rho), [\![\gamma]\!]\uparrow_c(\rho))$

The soundness theorem:

$$Theorem4 \in (M, N \in \Gamma \vdash A; M \cong N; \rho \in w \Vdash \Gamma)\ Eq([\![M]\!]\rho, [\![N]\!]\rho)$$

is shown mutually with a corresponding lemma for substitutions:

$$(\gamma, \gamma' \in \Gamma \to \Delta; \gamma \cong \gamma'; \rho \in w \Vdash \Gamma)\ Eq([\![\gamma]\!]\rho, [\![\gamma']\!]\rho)$$

they are both shown by induction on the rules of conversion. Notice that the soundness result holds in any Kripke model.

4.9. Decision algorithm for conversion

We now have a decision algorithm for testing convertibility between proof trees: compute the normal form and check if they are exactly the same. This decision algorithm is correct:

$$Theorem5 \in (\mathsf{Id}(nf(M), nf(N)))\ M \cong N$$

since by *Theorem2* we have $M \cong nf(M)$ and $N \cong nf(N)$ and, by hypothesis, $\mathsf{Id}(nf(M), nf(N))$, we get by transitivity of $\cong$, that $M \cong N$.

The decision algorithm is also complete

$$Theorem6 \in (M \cong N)\ \mathsf{Id}(nf(M), nf(N))$$

since by *Theorem4* and the hypothesis, $M \cong N$ we get $Eq([\![M]\!]id, [\![N]\!]id)$ and by *Corollary1* we get $\mathsf{Id}(nf(M), nf(N))$.

5. Normal form

As we have seen above it is not necessary to know that *nf* actually gives a proof tree in η-normal form for the results above. This is however the case. We can mutually inductively define when a proof tree is in long η-normal form, *enf*, and in applicative normal form, *anf*:

$$\frac{M \in \Gamma \vdash A}{anf(M) \in Set} \qquad \frac{M \in \Gamma \vdash A}{enf(M) \in Set}$$

The rules for them are:

$$\frac{M \in \Gamma \vdash o \quad anf(M)}{enf(M)} \qquad \frac{M \in [\Gamma, x : A] \vdash B \quad enf(M)}{enf(\lambda(x : A).M)}$$

$$\frac{}{anf(x^A_\Gamma)} \qquad \frac{M \in \Gamma \vdash A \to B \quad anf(M) \quad N \in \Gamma \vdash A \quad enf(N)}{anf(M\ N)}$$

We prove that $nf(M)$ is in long η-normal form. For this we define a Kripke logical predicate, $\mathcal{N}$, such that $\mathcal{N}([\![M]\!])$ and if $\mathcal{N}(u)$, then $enf(reify\,(u))$.

$$\frac{\Gamma \in \mathcal{C} \quad A \in \mathcal{T} \quad v \in \Gamma \Vdash A}{\mathcal{N}(v) \in Set}$$

For base type we intuitively define $\mathcal{N}(v)$ to hold if $anf(v)$.

$$\frac{v \in \Gamma \Vdash o \quad f \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma) anf(ground_c(v))}{\mathcal{N}(v)}$$

If $u \in \Gamma \Vdash A \to B$, then $\mathcal{N}(u)$ is defined to hold if $\mathcal{N}(app(u, v))$ holds for all $v \in \Gamma \Vdash A$ such that $\mathcal{N}(v)$.

$$\frac{\begin{array}{l} v \in \Gamma \Vdash A \to B \\ f \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma; u \in \Delta \Vdash A; \mathcal{N}(u)) \mathcal{N}(app_c(v, u)) \end{array}}{\mathcal{N}(v)}$$

We also define $\mathcal{N}(\rho)$, $\rho \in \Gamma \Vdash \Delta$, to hold if every value,v, in ρ has the property $\mathcal{N}(v)$. We prove the following lemmas:

- $(v \in \Delta \Vdash A; \mathcal{N}(v); c \in \Gamma \geq \Delta) \mathcal{N}(\uparrow_c(v))$
- $(\rho \in \Gamma \Vdash \Delta; \mathcal{N}(\rho); Occur(x, A, \Delta)) \mathcal{N}(lookup\,(\rho, x^A_\Delta))$
- $(\rho \in \Gamma_0 \Vdash \Delta; \mathcal{N}(\rho); c \in \Gamma_1 \geq \Gamma_0) \mathcal{N}(\uparrow_c(\rho))$
- $(\rho \in \Gamma \Vdash \Delta_1; \mathcal{N}(\rho); c \in \Delta_1 \geq \Delta_0) \mathcal{N}(\downarrow_c(\rho))$

which are used to prove that

$$Lemma10 \in (M \in \Gamma \vdash A; \rho \in \Gamma \Vdash \Delta; \mathcal{N}(\rho))\, \mathcal{N}([\![M]\!]\rho)$$

The lemma is proved together with a corresponding lemma for substitutions:

$$(\gamma \in \Delta \to \Gamma; \rho \in \Theta \Vdash \Delta; \mathcal{N}(\rho))\, \mathcal{N}([\![\gamma]\!]\rho)$$

The main lemma is that for all values, v, such that $\mathcal{N}(v)$, *reify* (v) returns a proof tree in η-normal form, which is intuitively proved together with a proof that for all proof trees in applicative normal form we can find a value, v, such that $\mathcal{N}(v)$. More precisely, the lemma

$$\textit{Lemma 11} \in (v \in \Gamma \Vdash A; \mathcal{N}(v))\textit{enf}(\textit{reify}\,(v))$$

is shown mutually with

$$(f \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma)\Delta \vdash A$$
$$; h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma)\textit{anf}(f(\Delta, c)))\, \mathcal{N}(\textit{val}_{\Gamma,A}(f))$$

The proofs are by induction on the types.

It is straightforward to prove that $\mathcal{N}(id)$ and then by *Lemma 11* and *Lemma 10* we get that $nf(M)$ is in long η-normal form, i.e.,

$$\textit{Theorem7} \in (M \in \Gamma \vdash A)\textit{enf}(nf(M))$$

Hence a proof tree is convertible with its normal form.

We can also use the results above to prove that if $\lambda(x:A).M \cong \lambda(y:A).N$, then $M(x=z) \cong N(y=z)$ where z is a fresh variable. Hence we have that λ is one-to-one up to α-conversion.

6. Application to terms

In practice we may not want to work with proof trees but rather well-typed terms. As an application of the results above we show how to give semantics to a formulation of Martin-Löf's substitution calculus [13, 20] in the simply typed setting. In this calculus we have a set of untyped terms, T, and we define when a term in T is well-typed and when two terms of a given type are convertible with each other.

In order to give semantics to untyped terms, we first define an erasure function that translates a proof tree M to an untyped term, denoted M^-. The main theorem is then to prove that if two proof trees M, N erase to the same term, $\mathsf{Id}(M^-, N^-)$, then $M \cong N$; it follows that M and N have the same semantics. For this we first prove that $\mathsf{Id}(nf(M)^-, nf(N)^-)$ implies $M \cong N$. We also define a reduction on the untyped terms $\Gamma \vdash t_1 \Downarrow t_2 : A$ that is deterministic (i.e., if $\Gamma \vdash t \Downarrow t_1 : A$ and $\Gamma \vdash t \Downarrow t_2 : A$, then $\mathsf{Id}(t_1, t_2)$) such that $\Gamma \vdash M^- \Downarrow nf(M)^- : A$. We then prove that if a proof tree M erases to a well-typed term t, then $t \Downarrow nf(M)^-$. Now, if two proof trees M and N erase to the same well-typed term t, then $t \Downarrow nf(M)^-$ and $t \Downarrow nf(N)^-$. Since the reduction is deterministic we have that $nf(M)^-$ and $nf(N)^-$ are the same, and hence $M \cong N$. The idea of this proof comes from Streicher [19] (chapter IV).

6.1. *Definition of terms*

We mutually define the set of terms, $\mathsf{T} \in \mathit{Set}$, and substitutions, $\mathsf{S} \in \mathit{Set}$ as

$$\frac{x \in \mathit{Name}}{\mathbf{v}(x) \in \mathsf{T}} \qquad \frac{x \in \mathit{Name} \quad t \in \mathsf{T}}{\mathbf{lam}(x, t) \in \mathsf{T}}$$

$$\frac{t_1 \in \mathsf{T} \quad t_2 \in \mathsf{T}}{\mathbf{app}(t_1, t_2) \in \mathsf{T}} \qquad \frac{s \in \mathsf{S} \quad t \in \mathsf{T}}{\mathbf{sub}(t, s) \in \mathsf{T}}$$

$$\frac{}{\mathbf{id} \in \mathsf{S}} \qquad \frac{s \in \mathsf{S} \quad x \in \mathit{Name} \quad t \in \mathsf{T}}{\mathbf{upd}(s, x, t) \in \mathsf{S}} \qquad \frac{s_1, s_2 \in \mathsf{S}}{\mathbf{com}(s_1, s_2) \in \mathsf{S}}$$

and use the following notational conventions:

x	for $\mathbf{v}(x)$
$\lambda x.t$	for $\mathbf{lam}(x, t)$
$t_1\ t_2$	for $\mathbf{app}(t_1, t_2)$
$t\ s$	for $\mathbf{sub}(t, s)$
$()$	for $\mathbf{id}$
$(s, x = t)$	for $\mathbf{upd}(s, x, t)$
$s_1 s_2$	for $\mathbf{com}(s_1, s_2)$

6.2. *Typing rules*

We give the typing rules for terms and substitutions mutually inductively

$$\frac{A \in \mathcal{T} \quad \Gamma \in \mathcal{C} \quad t \in \mathsf{T}}{\mathcal{TT}(\Gamma, A, t) \in \mathit{Set}} \qquad \frac{\Gamma, \Delta \in \mathcal{C} \quad s \in \mathsf{S}}{\mathcal{TS}(\Gamma, \Delta, s) \in \mathit{Set}}$$

We use the notational convention $\Gamma \vdash t : A$ for $\mathcal{TT}(\Gamma, A, t)$ and $\Gamma \vdash s : \Delta$ for $\mathcal{TS}(\Gamma, \Delta, s)$. We leave out the constructors in the presentation of the typing rules below.

$$\frac{\Gamma \geq \Delta \quad \Delta \vdash t : A}{\Gamma \vdash t : A} \qquad \frac{\mathit{Occur}(x, A, \Gamma)}{\Gamma \vdash x : A} \qquad \frac{[\Gamma, x : A] \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t_1 : A \to B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B} \qquad \frac{\Gamma \vdash s : \Delta \quad \Delta \vdash t : A}{\Gamma \vdash t\ s : A}$$

$$\frac{\Theta \geq \Gamma \quad \Gamma \vdash s : \Delta}{\Theta \vdash s : \Delta} \qquad \frac{\Delta \geq \Theta \quad \Gamma \vdash s : \Delta}{\Gamma \vdash s : \Theta}$$

$$\frac{}{\Gamma \vdash () : \Gamma} \qquad \frac{\Gamma \vdash s : \Delta \quad \Gamma \vdash t : A}{\Gamma \vdash (s, x = t) : [\Delta, x : A]} \qquad \frac{\Gamma \vdash s_1 : \Delta \quad \Delta \vdash s_2 : \Theta}{\Gamma \vdash s_2\ s_1 : \Theta}$$

6.3. Convertibility of terms

We mutually inductively define when two terms are convertible with each other together with the definition of convertibility between substitutions:

$$\frac{A \in \mathcal{T} \quad \Gamma \in \mathcal{C} \quad t_1, t_2 \in \mathsf{T}}{\mathcal{CT}(\Gamma, A, t_1, t_2) \in Set} \qquad \frac{\Gamma, \Delta \in \mathcal{C} \quad s_1, s_2 \in \mathsf{S}}{\mathcal{CS}(\Gamma, \Delta, s_1, s_2) \in Set}$$

We write $\Gamma \vdash t_1 \cong t_2 : A$ for $\mathcal{CT}(\Gamma, A, t_1, t_2)$ and $\Gamma \vdash t_1 \cong t_2 : \Delta$ for $\mathcal{CS}(\Gamma, \Delta, s_1, s_2)$. The rules are beside the symmetry, transitivity and congruence rules:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \cong t : A} \qquad \frac{\Gamma \vdash s : \Delta \quad [\Delta, x : A] \vdash t_1 : B \quad \Gamma \vdash t_2 : A}{\Gamma \vdash (\lambda x.t_1)s\ t_2 \cong t_1(s, x = t_2) : B}$$

$$\frac{T(\mathit{fresh}(x, \Gamma)) \quad \Gamma \vdash t : A \to B}{\Gamma \vdash t \cong \lambda x.(t\ x) : A \to B} \qquad \frac{\Gamma \geq \Delta \quad \Delta \vdash t_1 \cong t_2 : A}{\Gamma \vdash t_1 \cong t_2 : A}$$

$$\frac{T(\mathit{fresh}(x, \Delta)) \quad \Gamma \vdash s : \Delta \quad \Gamma \vdash t : A}{\Gamma \vdash x(s, x = t) \cong t : A} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash t() \cong t : A}$$

$$\frac{\Delta \vdash t_1 : A \to B \quad \Delta \vdash t_2 : A \quad \Gamma \vdash s : \Delta}{\Gamma \vdash (t_1\ t_2)s \cong (t_1 s)\ (t_2 s) : B}$$

$$\frac{\Gamma_2 \vdash s_1 : \Gamma_3 \quad \Gamma_1 \vdash s_2 : \Gamma_2 \quad \Gamma_3 \vdash t : A}{\Gamma_1 \vdash (t\ s_1)s_2 \cong t(s_1 s_2) : A}$$

$$\frac{\Gamma_1 \geq \Gamma_2 \quad \Gamma_2 \vdash s_1 \cong s_2 : \Delta}{\Gamma_1 \vdash s_1 \cong s_2 : \Delta} \qquad \frac{\Delta_1 \geq \Delta_2 \quad \Gamma \vdash s_1 \cong s_2 : \Delta_1}{\Gamma \vdash s_1 \cong s_2 : \Delta_2}$$

$$\frac{\Gamma \vdash s : \Delta}{\Gamma \vdash s() \cong s : \Delta} \qquad \frac{\Gamma_1 \vdash s_3 : \Gamma_2 \quad \Gamma_2 \vdash s_2 : \Gamma_3 \quad \Gamma_3 \vdash s_1 : \Delta}{\Gamma_3 \vdash (s_1 s_2)s_3 \cong s_1(s_2 s_3) : \Delta}$$

$$\frac{\Gamma_1 \vdash s_2 : \Gamma_2 \quad \Gamma_2 \vdash s_1 : \Delta \quad \Gamma_2 \vdash t : A}{\Gamma_1 \vdash (s_1, x = t)s_2 \cong (s_1 s_2, x = t\ s_2) : [\Delta, x : A]}$$

$$\frac{\Gamma \vdash s : []}{\Gamma \vdash s \cong () : []} \qquad \frac{\Gamma \vdash s : [\Delta, x : A]}{\Gamma \vdash s \cong (s, x = x\ s)}$$

$$\frac{T(\mathit{fresh}(x, \Delta)) \quad \Gamma \vdash s : \Delta \quad \Delta \vdash t : A}{\Gamma \vdash (s, x = t) \cong s : \Delta} \qquad \frac{\Gamma \vdash s : \Delta}{\Gamma \vdash ()s \cong s : \Delta}$$

It is straightforward to prove that if two terms (substitutions) are convertible with each other, then they are also well-typed.

7. Correspondence between proof trees and terms

We define a function that translates the proof trees to the corresponding untyped terms and likewise for the substitutions, we write M^- and γ^- for these operations. The definitions

are:

$$\begin{aligned}
\left(x^A_\Gamma\right)^- &= x\\
(\lambda(x:A).M)^- &= \lambda x.M^-\\
(M\ N)^- &= M^-\ N^-\\
(M\gamma)^- &= M^-\gamma^-\\
\pi_c^- &= ()\\
(\gamma, x = M)^- &= (\gamma^-, x = M^-)\\
(\delta\gamma)^- &= \delta^-\gamma^-
\end{aligned}$$

It is easy to prove that the translation of a proof tree is well-typed:

$$Lemma12 \in (M \in \Gamma \vdash A)\Gamma \vdash M^- : A$$

In general, we may have $\mathsf{Id}(M^-, N^-)$ but M different from N. Take for example $(\lambda(y : B\to B).z)\ \lambda(x : B).x \in [z : A] \vdash A$ and $(\lambda(y : C \to C).z)\ \lambda(x : C).x \in [z : A] \vdash A$ which are both translated into $(\lambda y.z)\ \lambda x.x$. This shows that a given term can be decorated into different proof trees.

We define a relation between terms and their possible decorations (and likewise for the substitutions) as an inductively defined set:

$$\frac{A \in \mathcal{T} \quad \Gamma \in \mathcal{C} \quad t \in \mathsf{T} \quad M \in \Gamma \vdash A}{Decorate(A, \Gamma, t, M) \in Set}$$

and

$$\frac{\Gamma, \Delta \in \mathcal{C} \quad s \in \mathsf{S} \quad \gamma \in \Gamma\to\Delta}{DecorateS(\Gamma, \Delta, s, \gamma) \in Set}$$

Instead of $Decorate(A, \Gamma, t, M)$ we write $t\ \mathcal{D}_{\Gamma,A}\ M$ or even $t\ \mathcal{D}\ M$ when the typing of M is clear. We write correspondingly for the substitutions. The introduction rules are (leaving out the constructors):

$$\frac{Occur(x, A, \Gamma)}{x\ \mathcal{D}\ x^A_\Gamma} \qquad \frac{t_1\ \mathcal{D}_{\Gamma,A\to B}\ M \quad t_2\ \mathcal{D}_{\Gamma,A}\ N}{t_1\ t_2\ \mathcal{D}_{\Gamma,B}\ M\ N}$$

$$\frac{c \in \Gamma \geq \Delta \quad t\ \mathcal{D}_{\Delta,A}\ M}{t\ \mathcal{D}_{\Gamma,A}\ M\pi_c} \qquad \frac{s\ \mathcal{D}_{\Gamma,\Delta}\ \gamma \quad t\ \mathcal{D}_{\Delta,A}\ M}{t\ s\ \mathcal{D}_{\Gamma,A}\ M\gamma}$$

$$\frac{t\ \mathcal{D}_{[\Gamma,x:A],B}\ M}{\lambda x.t\ \mathcal{D}_{\Gamma,A\to B}\ \lambda(x : A).M} \qquad \frac{c \in \Gamma \geq \Delta}{()\ \mathcal{D}_{\Gamma,\Delta}\ \pi_c}$$

$$\frac{s\ \mathcal{D}_{\Delta,\Gamma}\ \gamma \quad t\ \mathcal{D}_{\Delta,A}\ M}{(s, x = t)\ \mathcal{D}_{\Delta,[\Gamma,x:a]}\ (\gamma, x = M)} \qquad \frac{c \in \Gamma \geq \Delta \quad s\ \mathcal{D}_{\Theta,\Gamma}\ \gamma}{s\ \mathcal{D}_{\Theta,\Delta}\ \pi_c\gamma}$$

$$\frac{c \in \Theta \geq \Gamma \quad s\ \mathcal{D}_{\Gamma,\Delta}\ \gamma}{s\ \mathcal{D}_{\Theta,\Delta}\ \gamma\pi_c} \qquad \frac{s_1\ \mathcal{D}_{\Theta,\Gamma}\ \gamma_1 \quad s_2\ \mathcal{D}_{\Gamma,\Delta}\ \gamma_2}{s_1 s_2\ \mathcal{D}_{\Theta,\Delta}\ \gamma_1\gamma_2}$$

It is straightforward to prove that

$$Lemma13 \in (M \in \Gamma \vdash A)M^{-} \; \mathcal{D} \; M$$

mutually with a corresponding lemma for substitutions.

Using the discussion in Section 3.3 on how to define the monotonicity and projection rules with π we can find a proof tree that corresponds to a well-typed term:

$$Lemma14 \in (\Gamma \vdash t : A)\Sigma(M : \Gamma \vdash A).\mathsf{Id}(M^{-}, t)$$

As a direct consequence of this lemma and *Lemma13* we know that every well-typed term has a decoration

$$Lemma15 \in (\Gamma \vdash t : A)\Sigma(M : \Gamma \vdash A).t \; \mathcal{D} \; M$$

As a consequence of this lemma we can now define the semantics of a well-typed term in a Kripke model as the semantics of the decorated term. In the remaining text, however, we study only the correspondence between terms and proof trees since the translation to the semantics is direct.

As we mentioned above a well-typed term may be decorated to several proof trees. We can however prove that if two proof trees are in η-normal form and they are decorations of the same term, then the two proof trees are convertible. We prove:

$$Lemma16 \in (M, N \in \Gamma \vdash A; enf(M); enf(N); t \; \mathcal{D} \; M; t \; \mathcal{D} \; N)\mathsf{Id}(M, N)$$

together with two corresponding lemmas for proof trees in applicative normal form:

– $(M \in \Gamma \vdash A_1; N \in \Gamma \vdash A_2; anf(M); anf(N); t \; \mathcal{D} \; M; t \; \mathcal{D} \; N$
$)\mathsf{Id}(A_1, A_2)$
– $(M \in \Gamma \vdash A; N \in \Gamma \vdash B; anf(M); anf(N); t \; \mathcal{D} \; M; t \; \mathcal{D} \; N$
$)\mathsf{Id}(M, N)$

As a consequence we get that if $nf(M)^{-}$ and $nf(N)^{-}$ are the same, then $M \cong N$:

$$Corollary2 \in (M, N \in \Gamma \vdash A; \mathsf{Id}(nf(M)^{-}, nf(N)^{-}))) \; M \cong N$$

Proof: By *Lemma16* and *Theorem7* we get $\mathsf{Id}(nf(N), nf(M)$ and by *Theorem5* we get $M \cong N$. □

7.1. Reduction

We mutually inductively define when a term is in weak head normal form (abbreviated whnf) and in weak head applicative normal form (abbreviated whanf) by:

– $\lambda x.t$ is in whnf if t is in whnf,
– t is in whnf if t is in whanf,

– x is in whanf
– $t_1\ t_2$ is in whanf if t_1 is in whanf and t_2 is in whnf

We inductively define a deterministic untyped one-step reduction on terms and substitution (we use the symbol $\rightarrow$ for both):

– $(\lambda x.t)s\ a \rightarrow t\ (s, x = a)$
– If $t_1 \rightarrow t_2$, then $t_1\ t \rightarrow t_2\ t$
– $x\ (s, x = t) \rightarrow t$
– If $x \neq y$, then $x\ (s, y = t) \rightarrow x\ s$
– $x\ () \rightarrow x$
– If $s_1 \rightarrow s_2$, then $x\ s_1 \rightarrow x\ s_2$
– $(t_1\ t_2)s \rightarrow (t_1 s)\ (t_2 s)$
– $(t\ s_1)s_2 \rightarrow t\ (s_1 s_2)$
– $(s_0, x = t)s_1 \rightarrow (s_0 s_1, x = t\ s_1)$
– $(s_1 s_2)s_3 \rightarrow s_1(s_2 s_3)$
– $()s \rightarrow s$

The untyped evaluation to whnf , $\Rightarrow$, is inductively defined by:

– If t is in whnf, then $t \Rightarrow t$,
– if $t_1 \rightarrow t_2$ and $t_2 \Rightarrow t_3$, then $t_1 \Rightarrow t_3$.

It is easy to see that this relation is deterministic.

In order to define a deterministic reduction that gives a term on long η-normal form we need to use its type. We define this typed reduction, $\downarrow$, simultaneously with $\downarrow_s$ which η-expands the arguments in an application on whnf:

– $\Gamma \vdash t_0 \downarrow t_2 : o$, if there exist a term t_1 such that $t_0 \Rightarrow t_1$ and $\Gamma \vdash t_1 \downarrow_s t_2 : o$
– $\Gamma \vdash t_1 \downarrow \lambda z.t_2 : A \rightarrow B$, if $[\Gamma, z : A] \vdash t_1\ z \downarrow t_2 : B$ (where $z = gensym(\Gamma)$)
– $\Gamma \vdash x \downarrow_s x : A$, if $Occur(x, A, \Gamma)$
– $\Gamma \vdash t_1\ t_2 \downarrow_s t_1'\ t_2' : B$, if there exist A such that $\Gamma \vdash t_1 \downarrow_s t_1' : A \rightarrow B$ and $\Gamma \vdash t_2 \downarrow t_2' : A$

Finally we define $\Gamma \vdash t \Downarrow t' : A$ to hold if $\Gamma \vdash t\ () \downarrow t' : A$

7.2. *Equivalence between proof trees and terms*

We can prove that if $M \in \Gamma \vdash A$, then $\Gamma \vdash M^- \Downarrow nf(M)^- : A$. This we do by defining a Kripke logical relation

$$\frac{A \in \mathcal{T} \quad \Gamma \in \mathcal{C} \quad t \in \mathsf{T} \quad v \in \Gamma \Vdash A}{Redval(A, \Gamma, t, v)Set}$$

We write $t\ \mathcal{R}\ v$ instead of $Redval(A, \Gamma, t, v)$.

When $v \in \Gamma \Vdash o$ we intuitively have that $t \mathcal{R} v$ holds if $\Gamma \vdash t \downarrow v^{-}$.

$$\frac{\begin{array}{l} t \in \mathsf{T} \\ v \in \Delta \Vdash o \\ h \in (\Gamma \in \mathcal{C}; c \in \Gamma \geq \Delta; t' \in \mathsf{T}; t' \mathcal{D}\ ground_c(v))\Gamma \vdash t \downarrow t' : o \end{array}}{t \mathcal{R} v}$$

When $v \in \Gamma \Vdash A \to B$, then $t \mathcal{R} v$ holds if for all t' and $u \in \Gamma \Vdash A$ such that $t' \mathcal{R} u$, we have that $t\ t' \mathcal{R}\ app(v, u)$

$$\frac{\begin{array}{l} t \in \mathsf{T} \\ v \in \Delta \Vdash A \to B \\ h \in (\Gamma \in \mathcal{C}; c \in \Gamma \geq \Delta; u \in \Gamma \Vdash A; t' \in \mathsf{T}; \Gamma \vdash t' : A; t' \mathcal{R} u \\ \quad)\ (t\ t')\ \mathcal{R}\ (app_c(v, u)) \end{array}}{t \mathcal{R} v}$$

For the substitutions we define correspondingly:

$$\frac{\Gamma, \Delta \in \mathcal{C} \quad s \in \mathsf{S} \quad \rho \in \Gamma \Vdash \Delta}{Redenv(\Gamma, \Delta, s, \rho) Set}$$

We write $s \mathcal{R} \rho$ instead of $Redenv(\Gamma, \Delta, s, \rho)$. The introduction rules for $\mathcal{R}$ are (leaving out the constructors):

$$\frac{s \in \mathsf{S} \quad \Delta \vdash s : []}{s \mathcal{R} \{\}}$$

and

$$\frac{\begin{array}{l} s \in \mathsf{S} \\ \Delta \vdash s : [\Gamma, x : A] \\ \rho \in \Gamma \Vdash \Delta \\ v \in \Delta \Vdash A \\ s \mathcal{R} \rho \\ x\ s \mathcal{R} v \end{array}}{s \mathcal{R} \{\rho, x = v\}}$$

The following lemmas are straightforward to prove:

- $(v \in \gamma \Vdash A; t_1 \mathcal{R} v; t_2 \to t_1)\ t_2 \mathcal{R} v$
- $(\rho \in \Gamma \Vdash \Delta; \Delta \vdash s_1 : \Gamma; s_1 \to s_2; s_2 \mathcal{R} \rho)\ s_1 \mathcal{R} \rho$
- $(\rho \in \Gamma \Vdash \Delta; Occur(x, A, \Gamma); \Delta \vdash s : \Gamma)\ (x\ s) \mathcal{R}\ lookup\left(\rho, x^A_\Delta\right)$
- $(c \in \Gamma \geq \Delta; u \in \Delta \Vdash A; t \mathcal{R} u)\ t \mathcal{R} \uparrow_c (u)$

– $(c \in \Theta \geq \Delta; \Delta \vdash s : \Gamma; \rho \in \Gamma \Vdash \Delta; s \mathcal{R} \rho)\ s \mathcal{R} \uparrow_c (\rho)$
– $(c \in \Gamma \geq \Theta; \Delta \vdash s : \Gamma; \rho \in \Gamma \Vdash \Delta; s \mathcal{R} \rho)\ s \mathcal{R} \downarrow_c (\rho)$
– $(\Gamma \vdash t : A; \Gamma \vdash s : \Delta; \rho \in \Gamma \Vdash \Delta; s \mathcal{R} \rho)\ (s, x = t) \mathcal{R} \rho$

Using these lemmas we can prove by mutual induction on the proof tree of terms and substitutions that:

– $(M \in \Gamma \vdash A; \rho \in \Gamma \Vdash \Delta; \Delta \vdash s : \Gamma; t \mathcal{D} M; s \mathcal{R} \rho)\ t\ s \mathcal{R} [\![M]\!]\rho$
– $(\gamma \in \Gamma \vdash \Theta; \rho \in \Gamma \Vdash \Delta; \Delta \vdash s_2 : \Gamma; s_1 \mathcal{D} \gamma; s_2 \mathcal{R} \rho)\ (s_2\ s_1) \mathcal{R} [\![\gamma]\!]\rho$

We also show, intuitively, that if $t \mathcal{R} u$, $u \in \Gamma \Vdash A$, then $\Gamma \vdash t \downarrow \mathit{reify}(u)^- : A$:

$$\mathit{Lemma17} \in (\Gamma \vdash t_0 : A; u \in \Gamma \Vdash A; t_0 \mathcal{R} u; t_1 \mathcal{D} \mathit{reify}(u)) \Gamma \vdash t_0 \downarrow t_1 : A$$

together with a corresponding lemma for *val*:

$$(\Gamma \vdash t_0 : A; \mathit{whanf}(t_0); f \in (\Delta; c \in \Delta \geq \Gamma)\Delta \vdash A ; h \in (\Delta \in \mathcal{C}; c \in \Delta \geq \Gamma)\Delta \vdash t_0 \downarrow_s f(\Delta, c)^- : A)\ t \mathcal{R} \mathit{val}(f)$$

and the proof that the translation of proof trees reduces to the translation of its normal form follows directly:

$$\mathit{Theorem8} \in (M \in \Gamma \vdash A; t \mathcal{D} M)\Gamma \vdash t \Downarrow \mathit{nf}(M)^- : A$$

As a consequence we get that if two proof trees are decorations of the same term, then they are convertible with each other:

$$\mathit{Corollary3} \in (M, N \in \Gamma \vdash A; t \mathcal{D} M; t \mathcal{D} N))M \cong N$$

Proof: By *Theorem8* we get that $\Gamma \vdash t \Downarrow \mathit{nf}(M)^- : A$ and $\Gamma \vdash t \Downarrow \mathit{nf}(N)^- : A$. Since the reduction is deterministic we get $\mathsf{Id}(\mathit{nf}(M)^-, \mathit{nf}(N)^-)$ and by *Corollary2* we get that $M \cong N$. □

8. Correctness of conversion between terms

The conversion rules for terms are sound:

$$\mathit{Theorem9} \in (M, N \in \Gamma \vdash A; t_0 \mathcal{D} M, t_1 \mathcal{D} N; \Gamma \vdash t_0 \cong t_1 : A)M \cong N$$

Proof: The proof is by induction on the proof of $\Gamma \vdash t_0 \cong t_1 : A$. We illustrate the proof with the reflexivity case. In this case we have that t_0 and t_1 are the same, hence by *Corollary3* we get $M \cong N$.

To prove that the conversion rules are complete:

$$\mathit{Theorem10} \in (M, N \in \Gamma \vdash A; M \cong N)\Gamma \vdash M^- \cong N^- : A$$

is straightforward by induction on the proof of $M \cong N$. □

9. A decision algorithm for terms

The reduction defined above can be used for deciding if two well-typed terms are convertible with each other or not: reduce the terms and check if the results are equal. This algorithm is implicit in Martin-Löf's notes [13].

The decision algorithm is complete:

$$\mathit{Theorem11} \in (\Gamma \vdash t_0 \cong t_1 : A)\Sigma(t \in \mathsf{T}).\Gamma \vdash t_0 \Downarrow t \;\&\; \Gamma \vdash t_1 \Downarrow t$$

Proof: By *Lemma14* and *Lemma12* we know that there exist proof trees M, N such that t_0 is equal M^- and t_1 is equal to N^-. By *Theorem9* we get that $M \cong N$. We can choose $nf(M)^-$ for t since, by *Lemma8*, $\Gamma \vdash M^- \Downarrow nf(M)^- : A$, $\Gamma \vdash N^- \Downarrow nf(N)^- : A$ and by *Theorem6* we have $\mathsf{Id}((nf(M), nf(N))$ and hence that $nf(M)^-$ and $nf(N)^-$ are the same. □

This decision algorithm is correct, i.e.,

$$\mathit{Theorem12} \in (M, N \in \Gamma \vdash A; \Gamma \vdash M^- \Downarrow t : A; \Gamma \vdash N^- \Downarrow t : A)\; M \cong N$$

Proof: We have $\mathsf{Id}(nf(M)^-, nf(N)^-)$ since, by *Lemma8*, $\Gamma \vdash M^- \Downarrow nf(M)^- : A$ and $\Gamma \vdash N^- \Downarrow nf(N)^- : A$ and the reduction is deterministic. By *Corollary2* we get $M \cong N$. □

10. Conclusions

We have defined a calculus of proof trees for simply typed λ-calculus with explicit substitutions and we have proved that this calculus is sound and complete with respect to Kripke models. A decision algorithm for convertibility based on the technique "normalisation by evaluation" has also been proven correct.

One application of the results for proof trees is the soundness and completeness for a formulation of an implicitly-typed λ-calculus with explicit substitutions.

An important aspect of this work is that it has been carried out on a machine; actually, the problem was partly chosen because it seemed possible to do the formalization in a nice way using ALF. It is often emphasised that a proof is machine checked, but this is the very minimum one can 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: this work was not done first by pen and paper and then typed into the machine, but was from the very beginning carried out in interaction with the system.

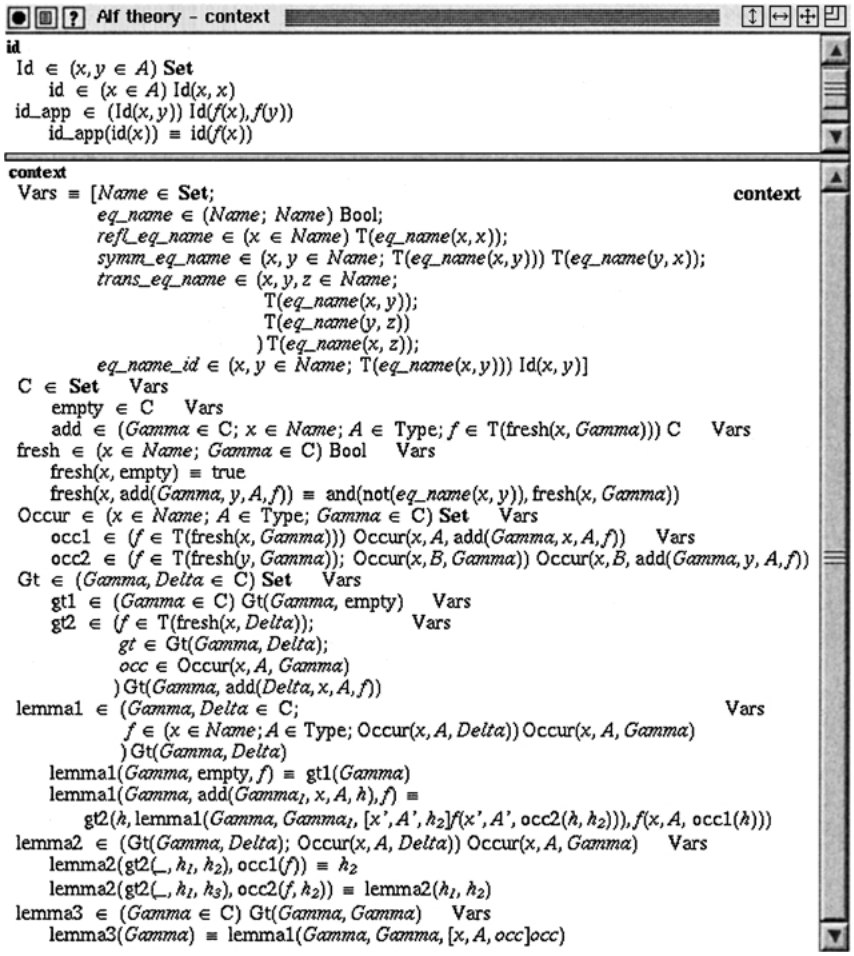

Figure 1. Snapshot from ALF's graphical interface.

Appendix

We present the proof as it is saved in text format and snapshot (Figure 1). The letter: corresponds to $\in$ in the text above and `Type` corresponds to $\mathcal{T}$.

```
Id : (A:Set;x,y:A)Set      []
  i : (A:Set;x:A)Id(A,x,x)      []
id_app : (A,B:Set;x,y:A;f:(A)B;Id(A,x,y)
          )Id(B,f(x),f(y))      []
         id_app(A,B,_,_,f,i(_,x)) = i(B,f(x))
```

```
Vars is [Name:Set;
         eq_name:(Name;Name)Bool;
         refl_eq_name:(x:Name)T(eq_name(x,x));
         symm_eq_name:(x,y:Name;T(eq_name(x,y))
                     )T(eq_name(y,x));
         trans_eq_name:(x,y,z:Name;T(eq_name(x,y))
                       ;T(eq_name(y,z)))T(eq_name(x,z));
         eq_name_id:(x,y:Name;T(eq_name(x,y))
                    )Id(Name,x,y)]

C : Set     Vars
fresh : (x:Name;Gamma:C)Bool     Vars
  empty : C     Vars
  add : (Gam:C;x:Name;A:Type;f:T(fresh(x,Gam)))C     Vars
fresh(x,empty) = true
fresh(x,add(Gamma,y,A,f))
      = and(not(eq_name(x,y)),fresh(x,Gamma))

Occur : (x:Name;A:Type;Gamma:C)Set     Vars
  occ1 : (x:Name;Gamma:C;A:Type;f:T(fresh(x,Gamma))
       )Occur(x,A,add(Gamma,x,A,f))     Vars
  occ2 : (x,y:Name;Gamma:C;A,B:Type;f:T(fresh(y,Gamma))
        ;Occur(x,B,Gamma)
       )Occur(x,B,add(Gamma,y,A,f))     Vars

Gt : (Gamma,Delta:C)Set     Vars
  gt1 : (Gamma:C)Gt(Gamma,empty)     Vars
  gt2 : (Gamma,Delta:C;x:Name;A:Type;f:T(fresh(x,Delta))
      ;gt:Gt(Gamma,Delta);occ:Occur(x,A,Gamma)
      )Gt(Gamma,add(Delta,x,A,f))     Vars

lemma1 : (Gamma,Delta:C;f:(x:Name;A:Type;Occur(x,A,Delta)
         )Occur(x,A,Gamma))Gt(Gamma,Delta)     Vars
lemma1(Gamma,empty,f) = gt1(Gamma)
lemma1(Gamma,add(Gamma1,x,A,h),f)
     = gt2(Gamma,Gamma1,x,A,h,lemma1(Gamma,Gamma1,
           [x',A',h2]f(x',A',occ2(x',x,Gamma1,A,A',h,h2)))
           ,f(x,A,occ1(x,Gamma1,A,h)))

lemma2 : (x:Name;A:Type;Gamma,Delta:C;Gt(Gamma,Delta)
          ;Occur(x,A,Delta))Occur(x,A,Gamma)     Vars
lemma2(x,A,Gamma,_,gt2(_,_,_,_,_,h1,h2)
      ,occ1(_,Gamma1,_,f))
     = h2
```

```
lemma2(x,A,Gamma,_,gt2(_,_,_,_,_,h1,h3)
      ,occ2(_,y,Gamma1,A1,_,f,h2))
     =  lemma2(x,A,Gamma,Gamma1,h1,h2)

lemma3 : (Gamma:C)Gt(Gamma,Gamma)     Vars
lemma3(Gamma) = lemma1(Gamma,Gamma,[x,A,occ]occ)
```

Acknowledgment

The author wants to thank the editor Olivier Danvy for having had the patience with delays and still being supportive. She is also grateful to Bernd Grobauer for help with improving the presentation. The comments of the anonymous referees have also been very valuable.

References

1. Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. Explicit substitutions. *Journal of Functional Programming*, **1**(4) (1991) 375–416.
2. Berger, U. Program extraction from normalization proofs. In *Proceedings of TLCA'93*, LNCS, Vol. 664, Springer Verlag, Berlin, 1993, pp. 91–106.
3. Berger, U. and Schwichtenberg, H. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, 1991, pp. 203–211.
4. Coquand, Th. Pattern matching with dependent types. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, B. Nordström, K. Petersson, and G. Plotkin (Eds.). Dept. of Computing Science, Chalmers Univ. of Technology and Göteborg University. Available at http://www.lfcs.informatics.ed.ac.uk/research/typesbra/proc/index.html, pp. 66–79.
5. Coquand, Th. and Dybjer, P. Intuitionistic model constructions and normalisation proofs. *Math. Structures Comput. Sci.*, **7**(1) (1997) 75–94.
6. Coquand, Th. and Gallier, J. A proof of strong normalization for the theory of constructions using a Kripke-like interpretation. In *Proceedings of the First Workshop in Logical Frameworks*, G. Huet and G. Plotkin (Eds.). Available at http://www.lfcs.informatics.ed.ac.uk/research/types-bra/proc/index.html, pp. 479–497.
7. Coquand, Th., Nordström, B., Smith, J., and von Sydow, B. Type theory and programming. *EATCS*, **52** (1994) 203–228.
8. Curien, P.-L. An abstract framework for environment machines. *Theoretical Computer Science*, **82** (1991) 389–402.
9. Friedman, H. Equality between functionals. In *Logic Colloquium, Symposium on Logic*, held at Boston, 1972–1973, LNCS, Vol. 453, Springer-Verlag, Berlin, 1975, pp. 22–37.
10. Gandy, R.O. On the axiom of extensionality—Part I. *The Journal of symbolic logic*, **21** (1956) 36–48.
11. Kripke, S.A. Semantical analysis of intuitionistic logic I. In *Formal Systems and Recursive Functions*, J.N. Crossley and M.A.E. Dummet (Eds.). North-Holland, Amsterdam, 1965, pp. 92–130.
12. Magnusson, L. and Nordström, B. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, Vol. 806, Springer-Verlag, Berlin, 1994, pp. 213–237.
13. Martin-Löf, P. Substitution calculus. Handwritten notes, Göteborg, 1992.
14. Mitchell, J.C. Type systems for programming languages. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen (Ed.). Elsevier and MIT Press, Amsterdam, 1990, pp. 365–458.
15. Mitchell, J.C. and Moggi, E. Kripke-style models for typed lambda calculus. *Annals for Pure and Applied Logic*, **51** (1991) 99–124.

16. Nordström, B., Petersson, K., and Smith, J. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, Oxford, UK, 1990.
17. Scott, D.S. Relating theories lambda calculus. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, New York, 1980, pp. 403–450.
18. Statman, R. Logical relation and the typed λ-calculus. *Information and Control*, **65** (1985) 85–97.
19. Streicher, T. *Semantics of Type Theory.* Birkhäuser, Basel, 1991.
20. Tasistro, A. Formulation of Martin-Löf's theory of types with explicit substitutions. Licentiate thesis, Department of Computing Science, Chalmers Univ. of Technology and University of Göteborg, 1993.