A Formalization of the Simply Typed Lambda Calculus in Coq

Adam Koprowski

Eindhoven University of Technology
Department of Computer Science
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
A.Koprowski@tue.nl

Abstract. In this paper we present a formalization of the simply typed lambda calculus with constants and with typing à la Church. It has been accomplished using the theorem prover Coq. The formalization includes, among other results, definitions of typed terms over arbitrary many-sorted signature, a substitution operating on typing judgements, an equivalence relation generalizing the concept of α-convertibility to free variables and a proof of strong normalization of β-reduction. The formalization has been used in a bigger project of certification of the higher-order recursive path ordering where well-foundedness of the union of the higher-order recursive path ordering and β-reduction was the main result.

1 Introduction

Simply typed lambda calculus ($\lambda^{\rightarrow}$), introduced by Church in 1940 [9], is a well-known computational model and a base for typed functional programming languages, such as ML and Haskell. The extensions of lambda calculi with dependent types are the base of intuitionistic type theory and the calculus of constructions and hence form the theoretical foundations for some theorem provers, including Coq.

In this paper we describe our accomplishment of a formalization of the simply typed lambda calculus with constants and with typing á la Church. The formalization uses de Bruijn notation for terms [8] and was carried out in Coq [17] – a theorem prover based on the calculus of inductive constructions. The formalization is complete (axiom-free) and fully constructive. The main results include:

(1) typed version of a variable substitution,
(2) notion of convertibility of terms extending the concept of α-convertibility to free variables and
(3) strong normalization for β-reduction.

It is worth noting here that our approach to defining substitution is different than a standard one. It is usual to formalize substitution operating on un-typed

terms and then argue that under some typing conditions on the inputs, a typing derivation for the result can be provided. Typically this argumentation is not formalized. However we want to use this formalization in the context of higher-order rewriting. There we want to verify that under some restrictions on rewrite rules the rewrite step, involving substitution, is well-typed (see [12, 13] for details). Hence we want to verify those typing conditions for substitution and this essentially comes down to defining substitution for typed terms. As we shall see in Section 4 this is not trivial even on paper, let alone in Coq.

This formalization has been used in the context of a bigger project of a formalization of the higher-order recursive path ordering (HORPO) [14] which in turn is part of the CoLoR project: a Coq library on rewriting and termination [1]. The goal of the CoLoR project is to formalize results from term rewriting and, ultimately, to certify termination proof candidates produced by tools designed for proving termination of term rewriting automatically. The HORPO formalization follows the paper presentation in [11] and partly [12]. The main result of this work is the strong normalization of the union of HORPO relation and the β-reduction of simply typed lambda calculus. Simply typed lambda calculus is used as a metalanguage in the considered variant of higher-order rewriting. Hence a formalization of HORPO requires a formalization of $\lambda^{\to}$, which was the main motivation for the work presented in this paper. For more information on the development of HORPO see [14, 13] and the project's web-site: `http://www.win.tue.nl/~akoprows/coq-horpo`.

It took the author many long months to complete the work on this formalization[1]. Only its part concerning $\lambda^{\to}$ contains almost 15,000 lines of Coq scripts and more than 400,000 characters. This makes it impossible to present all the aspects of this work here. Because the subject of strong normalization of $\lambda^{\to}$ has been extensively discussed in the literature and its formalization is the main subject of [7] in this paper we will focus on the other two of three main points we enumerated above: typed substitution and convertibility of terms. For more detailed description, all the proofs that we omit here and the discussion of this work in the context of the HORPO development, see [13].

The paper is organized as follows. We begin by discussion of related work in Section 2. Then in Section 3 we define terms of $\lambda^{\to}$. In Section 4 we introduce typed substitution. Section 5 discusses the convertibility relation on terms. Finally we conclude in Section 6. In every section we interleave the formal presentation with remarks on how these concepts are represented in Coq and with discussion of main difficulties that we encountered in their formalization.

2 Related work

Concerning related work at first place we should mention the work by Berger et al. In their recent work [7], independent of our formalization, they proved strong normalization of $\lambda^{\to}$ in three different theorem provers, including Coq[2] and from

[1] The whole formalization, including HORPO results, took two years of work.

[2] In fact only their Coq formalization is axiom-free.

those proofs machine-extracted implementations of normalization algorithms. In their Coq formalization they used, just as we do, terms in de Bruijn notation and typing á la Church and their normalization proof also relies on Tait computability predicate proof method, however their terms do not contain constants. The main difference between their formalization and the formalization presented in this paper is the fact that their prime goal was extraction of a certified normalization algorithm, whereas for us a somewhat more complete formalization of $\lambda^{\rightarrow}$ was required with the application to HORPO formalization in mind. Therefore work by Berger et al. addresses only the last out of the three main points of our formalization – strong normalization of β-reduction, which for us was not a goal in itself but rather a by-product of the main result of HORPO formalization. This also explains why our formalization, or rather only its part concerning $\lambda^{\rightarrow}$, is approximately four times bigger in terms of the size of Coq scripts.

Another important source of related work is the POPLMARK challenge [10]: a set of benchmarks for measuring progress in the area of formalizing metatheory of programming languages. Among numerous submissions there are even few using Coq and de Bruijn representation for terms. The comparison however is difficult: although the benchmark is designed for a richer type system (System $F_{<:}$) it focuses on completely different aspects and does not address any of the three main points presented in the introduction that are crucial for us.

Other somehow related formalizations include strong normalization proofs for calculi like: Calculus of Constructions [6], [3]; System F [2]; typed λ-calculus with co-products [4] and λ-calculi with weak reduction strategies [16].

3 Definition of Terms

In this section we introduce $\lambda^{\rightarrow}$ terms. For more comprehensive introduction to simply typed lambda calculus see, for instance, [5].

We assume a set of *ground types* $\mathcal{S}$ and *simple types*, $\mathcal{T}_{\mathcal{S}}$, are built from ground types using $\rightarrow$ constructor as usual. We will denote simple types by α, β etc. We also assume a set of *constants* $\mathcal{F}$ and define a *signature*, Σ, as a set of typed constants, that is pairs $f : \alpha$ with $f \in \mathcal{F}$ and $\alpha \in \mathcal{T}_{\mathcal{S}}$. Finally we assume a set of variables $\mathcal{V}$.

To represent terms in Coq we took advantage of Coq's module machinery. So firstly we defined the module `SimpleTypes` containing definition of simple types parameterized by a set of ground types.[3]

```
Module Type SimpleTypes.
  Parameter BaseType: Set.
  Parameter baseTypesNotEmpty: BaseType.
  Inductive SimpleType : Set :=
    | BasicType (T: BaseType)
```

[3] In some cases definitions presented here are somewhat simplified versions of the actual definitions used in the formalization. We encourage the reader to study the actual Coq scripts available at `http://www.win.tue.nl/~akoprows/coq-horpo`.

```
    | ArrowType (A B : SimpleType).
  Notation "x --> y" := (ArrowType x y)
    (at level 55, right associativity) : type_scope.
  Notation "# x " := (BasicType x) (at level 0) : type_scope.
End SimpleTypes.
```

Then the module Signature builds on that and contains definition of constants along with a function mapping them to their types.

```
Module Type Signature.
  Declare Module ST : SimpleTypes.
  Parameter FunctionSymbol: Set.
  Parameter f_type: FunctionSymbol -> SimpleType.
End Signature.
```

All further developments are contained within functors taking signature as their argument.

Now we define *environments* (often called typing contexts) as finite sets of typed variables, $\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ with $x_i \in \mathcal{V}$ such that for all $1 \leq i, j \leq n$, $x_i \in \mathcal{V}$, $\alpha_i \in \mathcal{T}_{\mathcal{S}}$ and $x_i \neq x_j$ for $i \neq j$. The domain of the environment is defined as $Var(\Gamma) = \{x_1, \ldots, x_n\}$.

In the formalization we use de Bruijn indices [8] to represent terms in order to avoid having to explicitly deal with α-conversion. So variables are natural numbers and environments can be represented as lists of simple types. However for the lifting operation required for substitution (see Section 4) we will need dummy variables. Therefore environments are lists of SimpleType option, and not SimpleType, with None representing dummy variables.

```
Definition Env := list (option SimpleType).
Definition EmptyEnv : Env := nil.
Definition VarD  E x A := nth_error E x = Some (Some A).
Definition VarUD E x   := nth_error E x = None \/
                          nth_error E x = Some None.
Notation "E |= x := A" := (VarD  E x A) (at level 60).
Notation "E |= x :!"   := (VarUD E x)   (at level 60).
Definition decl A E := Some A :: E.
Infix "[#]" := decl (at level 20, right associativity).
```

Note that the presence of dummy variables causes some complications as we loose the unique representation of environment: now both `nil` and `None::nil` represent an empty environment. One solution would be not to use dummy variables and incorporate the definition of lifting into the definition of substitution as this is the only place where we need to compute lifted versions of terms. However the definition of substitution is complicated enough even with lifting being defined separately. Moreover in the present version a number of properties of the lifting operation has been proven, indicating that having it as a separate entity was a good choice. Instead a definition of dedicated equality for environments,

not coinciding with Leibniz equality, and taking this problem into account has been provided. See [13] for details.

Now we define un-typed terms which we will call *preterms* by means of the following abstract syntax with variable, constant, application and abstraction respectively.

$$\mathcal{P}t := \mathcal{V} \mid \Sigma \mid @(\mathcal{P}t, \mathcal{P}t) \mid \lambda\mathcal{V}{:}\mathcal{T}_{\mathcal{S}}.\mathcal{P}t$$

Definition of preterms in Coq is rather straightforward. Again we introduce a number of notational conventions to make representation of terms more readable.

```
Inductive Preterm : Set :=
  | Var (x: nat)
  | Fun (f: FunctionSymbol)
  | Abs (A: SimpleType)(M: Preterm)
  | App (M N: Preterm).

Notation "^ f" := (Fun f) (at level 20).
Notation "% x" := (Var x) (at level 20).
Infix "@@" := App (at level 25, left associativity).
Notation "\ A => M" := (Abs A M) (at level 35).
```

Then *typing judgements* provide a typing discipline that terms must conform to. They are of the form $\Gamma \vdash t : \alpha$ denoting that in an environment Γ a preterm t has type α and are defined by the following inference rules:

$$\frac{x{:}\alpha \in \Gamma}{\Gamma \vdash x : \alpha}, \qquad \frac{f{:}\alpha \in \Sigma}{\Gamma \vdash f : \alpha},$$

$$\frac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash u : \alpha}{\Gamma \vdash @(t,u) : \beta}, \qquad \frac{\Gamma \cup \{x{:}\alpha\} \vdash t : \beta}{\Gamma \vdash \lambda x{:}\alpha.t : \alpha \to \beta}.$$

This inference system is easily recognizable in the following Coq definition:

```
Reserved Notation "E |- Pt := A" (at level 60).
Inductive Typing : Env -> Preterm -> SimpleType -> Set :=
| TVar: forall E x A, E |=  x := A -> E |- %x := A
| TFun: forall E f, E |- ^f := f_type f
| TAbs: forall E A B Pt, A [#] E |- Pt := B ->
          E |- \A => Pt := A --> B
| TApp: forall E A B PtL PtR, E |- PtL := A --> B ->
          E |- PtR := A -> E |- PtL @@ PtR := B
where "E |- Pt := A" := (Typing E Pt A).
```

Finally typed terms are built from an environment, a preterm, a type and a typing judgement certifying that in the given environment, the given preterm has the given type.

```
Record Term : Set := buildT {
  env:    Env;
```

```
  term:   Preterm;
  type:   SimleType;
  typing: Typing env term type
}.
```

Now we will present some results concerning the typing discipline for $\lambda^{\rightarrow}$. The first important results concern uniqueness of typing and type derivations.

Lemma 1. *Suppose* $\Gamma \vdash t : \alpha$ *and* $\Gamma \vdash t : \beta$ *then* $\alpha = \beta$.

Lemma 2. *Given term* $\Gamma \vdash t : \alpha$ *its type derivation is unique.*

The first result can easily be proven in Coq by structural induction on term.

```
Lemma Type_unique : forall Pt E T1 T2
  (D1 : Typing E Pt T1) (D2 : Typing E Pt T2), T1 = T2.
```

However the uniqueness of typing judgements is more subtle as it involves the usage of the uniqueness of identity proofs for dependent types and makes use of the Streicher's axiom K from the Coq standard library.

```
Lemma typing_unique : forall E Pt T (D1 D2 : Typing E Pt T),
  D1 = D2.
```

The easy consequence of the above results is that two terms with equal environments and equal structure are equal.

Lemma 3. *Given terms* $\Gamma \vdash t : \alpha$ *and* $\Delta \vdash u : \beta$, *if* $\Gamma = \Delta$ *and* $t = u$ *then* $\Gamma \vdash t : \alpha = \Delta \vdash u : \beta$.

with its Coq counterpart:

```
Lemma term_eq: forall M N,
  env M = env N -> term M = term N -> M = N.
```

This result will play a crucial role throughout the development. For any reasoning involving showing equality of two terms it allows to split the argument to showing equality of terms structure, so equality of un-typed terms, and proving that environments are equal.

Finally we prove that typing problem, that is given environment Γ and preterm t whether there exists a type α such that $\Gamma \vdash t : \alpha$, is decidable. Due to constructive nature of this proof a certified algorithm for type synthesis can be extracted from the proof.

```
Theorem autoType E Pt : {N: Term | env N = E & term N = Pt} +
  {~exists N: Term, env N = E /\ term N = Pt}.
```

Definitions introduced in this section are very crucial as we will constantly work with them in the course of formalization. We conclude this section with presentation of an alternative definition of terms and discussion of why it has not been used.

The alternative definition of terms uses a single inductive definition that combines the term structure with its typing judgement as follows:

```
Inductive TTerm : Env -> SimpleType -> Set :=
| TVar: forall E x A, E |= x := A -> TTerm E A
| TFun: forall E f, TTerm E (f_type f)
| TAbs: forall E A B, TTerm (A [#] E) B -> TTerm E (A --> B)
| TApp: forall E A B, TTerm E (A --> B) -> TTerm E A -> TTerm E B.
```

At first sight this definition looks very attractive and indeed has some advantages. The main drawback, however, is the fact that a term structure is embedded in this single inductive type and any reasoning about terms involves working with this complex definition, whereas with our first definition, thanks to `term_eq`, many proofs concerning terms require only dealing with two very simple definitions: Env and Preterm, as explained above. This was the main reason for preference of `Term` over `TTerm`. Another was that the development contains many results concerning un-typed terms which calls for making them separate entities. Making them the building blocks of terms rather than extracting them from the definition of `TTerm` seemed to be a more appropriate choice.

4 Substitution

In this section we present a substitution operating on typed terms. This means that the entities being substituted are typing judgements. We follow the presentation from [12]. The part concerning substitution is by far the largest part of the whole development.

While applying substitution to a term, substituted terms are put within some context, possibly within a context of some binders so, to avoid unintentional capturing of variables, their de Bruijn indices need to be increased. This operation is called lifting and we start by giving its precise definition. Note that for the definition of the lifting operation we explicitly use the fact that terms are represented using de Bruijn notation, so $\mathcal{V} = \mathbb{N}$.

For a term t and $n, k \in \mathbb{N}$ we define its lifted version, $t\uparrow_k^n$, with variables less than k untouched and remaining ones increased by n as: (with $t\uparrow^n \equiv t\uparrow_0^n$)

$$\begin{aligned} f\uparrow_k^n &= f, \\ x\uparrow_k^n &= x, && \text{if } x < k, \\ x\uparrow_k^n &= x + n, && \text{if } x \geq k, \\ @(t_l, t_r)\uparrow_k^n &= @(t_l\uparrow_k^n, t_r\uparrow_k^n), \\ \lambda x{:}\alpha.t\uparrow_k^n &= \lambda x{:}\alpha.t\uparrow_{k+1}^n . \end{aligned}$$

```
Fixpoint prelift_aux (n: nat) (P: Preterm) (k: nat)
 {struct P} : Preterm :=
  match P with
  | Fun _ => P
  | Var i => match (le_gt_dec k i) with
             | left _ =>  (* i >= k *) Var (i + n)
             | right _ => (* i < k *)  Var i
             end
```

```
  | App M N => App (prelift_aux n M k) (prelift_aux n N k)
  | Abs A M => Abs A (prelift_aux n M (S k))
  end.
Definition prelift P n := prelift_aux n P 0.
```

We also define lifting of environments. Given an environment $\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ we define its lifted version, $\Gamma\uparrow_k^n$, as: (with $\Gamma\uparrow^n \equiv \Gamma\uparrow_0^n$)

$$\Gamma\uparrow_k^n := \{x_i : \alpha_i \mid x_i : \alpha_i \in \Gamma, k > i \in \mathbb{N}\} \cdot \{(x_i + n) : \alpha_i \mid x_i : \alpha_i \in \Gamma, k \leq i \in \mathbb{N}\}$$

For Coq definition of lifting of environments three auxiliary functions on lists are used:

- `initialSeg l n` returns first 'n' elements of a list 'l' (or less if list is shorter).
- `finalSeg l n` return the suffix of 'l' starting at position 'n'.
- `copy n el` returns a list containing 'n' copies of 'el'.

```
Definition liftedEnv (n: nat) (E: Env) (k: nat) : Env :=
  initialSeg E k ++ copy n None ++ finalSeg E k.
```

So 'n' dummy variables are inserted at position 'k' which has precisely the effect of increasing indices of all the variables bigger than 'k' by 'n'.

The following result ensures that lifted terms are well-typed.

Lemma 4. *If $\Gamma \vdash t : \alpha$ then for any $n, k \in \mathbb{N}$: $\Gamma\uparrow_k^n \vdash t\uparrow_k^n : \alpha$.*

In Coq writing explicitly the term (definition) computing lifted version of typed term would be a difficult task. Fortunately using specification of the result that is precise enough we can use the Coq proving machinery to help us interactively build such a term conforming to this specification.

```
Definition lift_aux (n: nat) (M: Term) (k: nat) :
  {N: Term | env N = liftedEnv n (env M) k /\
             term N = prelift_aux n (term M) k /\
             type N = type M }.
Proof. (...) Defined.
Definition lift (M: Term)(n: nat) : Term :=
  proj1_sig (lift_aux n M 0).
```

Now we can move on to the definition of substitution.

A *substitution* is a finite set of pairs of variables and typed terms:

$$\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \ldots, x_n/\Gamma_n \vdash t_n : \alpha_n\},$$

such that for all $i \neq j \in \{1, \ldots, n\}$, $x_i \neq x_j$. A *substitution domain* is defined as an environment: $Dom(\gamma) = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ and a *substitution range* as an environment: $Ran(\gamma) = \bigcup_{i \in \{1,\ldots,n\}} \Gamma_i$. Abusing notation we will also write $x \in Dom(\Gamma)$ for $x \in Var(Dom(\Gamma))$. By $\gamma_{\setminus\mathcal{X}}$ we denote a substitution γ with its domain restricted to $\mathcal{V} \setminus \mathcal{X}$, that is:

$$\gamma_{\setminus\mathcal{X}} = \{(x_i/\Gamma_i \vdash t_i : \alpha_i) \in \gamma \mid i \in \{1, \ldots, n\}, x_i \notin \mathcal{X}\}.$$

Because in the formalization we use de Bruijn indices, substitution is simply a list of `option Term` (`None` indicating that given index is not in the domain of given substitution).

```
Definition Subst := list (option Term).
```

The definitions of the substitution domain and range are as expected.

Then a substitution on preterms is defined as usual:

$$\begin{aligned} x\gamma &= x, && \text{if } x \notin \mathit{Dom}(\gamma), \\ x\gamma &= u, && \text{if } x/\Gamma \vdash u : \alpha \in \gamma, \\ f\gamma &= f, \\ @(t_l, t_r)\gamma &= @(t_l\gamma, t_r\gamma), \\ (\lambda x{:}\alpha.t)\gamma &= \lambda x{:}\alpha.t\gamma_{\setminus\{x\}}. \end{aligned}$$

Using de Bruijn notation upon encounter of a binder substitution is alerted from `G` to `None::G`. Putting `None` as the head of the list has the effect of removing x from the domain of a substitution just as taking $\gamma_{\setminus\{x\}}$ in the above definition. Because of the presence of the binder also indices of all free variables within abstraction body are increased by one hence `None::G` has precisely legitimate effect. When substituting for variables we also need to lift indices of substituted terms as they are put in the context of some binders. That is why in the following definition we count the number of abstractions in the context on l and while performing substitution for variable first we lift the substituted term by l.

```
Fixpoint presubst_aux (P: Preterm)(l: nat)(G: Subst)
   {struct P} : Preterm :=
  match P with
  | Fun _ => P
  | Var i => match (nth_error G i) with
             | Some (Some Q) => term (lift Q l)
             | _ => Var i
             end
  | App M N => App (presubst_aux M l G) (presubst_aux N l G)
  | Abs A M => Abs A (presubst_aux M (S l) (None::G) )
  end.
Definition presubst P G := presubst_aux P 0 G.
```

To present the requirements that a substitution must satisfy in order to be applicable to a term we first need to introduce some definitions concerning environments.

First we define two simple operations on environments. For environments Γ, Δ we define the binary operations of composition and subtraction of environments as:

- $\Gamma \cdot \Delta = \Delta \cup \{x{:}\alpha \in \Gamma \mid x \notin \mathit{Var}(\Delta)\}$.
- $\Gamma \setminus \Delta = \{x{:}\alpha \in \Gamma \mid x \notin \mathit{Var}(\Delta)\}$.

We will say that environments Γ and Δ are *compatible*, denoted as $\Gamma \leftrightsquigarrow \Delta$, iff for any variable v if they both declare it, they declare it with the same type.

$$\Gamma \leftrightsquigarrow \Delta \;\equiv\; \forall x \in \mathcal{V} \,.\, x{:}\alpha \in \Gamma \wedge x{:}\beta \in \Delta \implies \alpha = \beta.$$

Finally we can present the conditions required for a substitution to be applicable to a term, captured by a notion of compatibility. A substitution $\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \ldots, x_n/\Gamma_n \vdash t_n : \alpha_n\}$ is *compatible* with a term $\Gamma \vdash t : \alpha$ if the following conditions are satisfied:

- Environments of terms in γ are compatible:
  $\forall i \neq j \in \{1, \ldots, n\} \,.\, \Gamma_i \leftrightsquigarrow \Gamma_j$.
- The domain of γ is compatible with the environment of t:
  $\Gamma \leftrightsquigarrow \mathit{Dom}(\gamma)$.
- Declarations in the range of γ not present in the domain of γ are compatible with the environment of t:
  $\Gamma \leftrightsquigarrow \mathit{Ran}(\gamma) \setminus \mathit{Dom}(\gamma)$.

The Coq definitions of the above notions are straightforward with `M [+] N`, `M [-] N` and `M [<->] N` representing respectively composition, subtraction and compatibility of environments `M` and `N` and `correct_subst M G` being a proposition expressing compatibility of substitution `G` with term `M`. See [13] for details.

The following result ensures that the conditions required by the compatibility of a substitution with a term are sufficient to type check the result of substitution. This is a stronger version of the result from [12]. For a proof see [13].

Lemma 5. *Let $\Gamma \vdash t : \alpha$ be a term and let $\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \ldots, x_n/\Gamma_n \vdash t_n : \alpha_n\}$ be a substitution compatible with this term. Then:*

$$\Gamma \setminus \mathit{Dom}(\gamma) \cdot \mathit{Ran}(\gamma) \vdash t\gamma : \alpha.$$

The proof of the above lemma is the base of the formalization of the typed version of the substitution:

```
Definition subst_aux (M: Term) (G: Subst) (C: correct_subst M G)
  : {Q: Term | env Q  = env M [-] subst_dom G [+] subst_ran G /\
               term Q = presubst (term M) G /\
               type Q = type M }.
Proof. (...) Defined.
Definition subst (M: Term) (G: Subst)
 (C: correct_subst M G) : Term := proj1_sig (subst_aux C).
```

5 Convertibility of Terms

During the development of computability properties for Tait computability predicate proof method that were to be used for proving well-foundedness of the higher order recursive path ordering, it was necessary to abstract away from some properties of terms and work modulo a certain equivalence on terms, $\sim$. If $t \sim t'$ then we will say that t and t' are $\sim$-convertible. The requirements on this relation were as follows:

(i) it should extend α-convertibility, so for α-convertible terms $t =_\alpha u$ we want to have $t \sim u$,

(ii) it should relate terms that differ only on some additional declarations in environments that are not used, so we want to have $\Sigma \vdash t : \alpha \sim \Sigma' \vdash t : \alpha$ if $\Sigma \leftrightsquigarrow \Sigma'$,

(iii) finally we want it to relate terms that differ only on names of free variables that is we want to have $t \sim u$ if there exists a renaming of variables γ such that $t = u\gamma$. With that respect $\sim$ can be seen as a natural extension of α-convertibility concept to free variables.

Another important point is that to achieve our goal we did not want to alert the typing rules of the calculus in any way. Presenting motivation for the above requirements is beyond the scope of this paper, we refer the interested reader to [13]. First we will present a convertibility relation on terms $\sim$ satisfying posed requirements and at the end of the section we will describe its development in Coq.

The idea behind $\sim$ is, roughly speaking, to define two terms to be equivalent if there exists an endomorphism on free variables of one term that maps them to free variables of the other term.

A *variable mapping* is a partial, injective function $\Phi : \mathcal{V} \to \mathcal{V}$. Since Φ is injective there exists its inverse Φ^{-1} and since this symmetry will play an important role we will write variable mappings using infix notation so $x\ \Phi\ y$ instead of $\Phi(x) = y$.

Now given such variable mapping Φ we say that *environments Γ and Δ are convertible* modulo this mapping, denoted as: $\Gamma \overset{\Phi}{\approx} \Delta$, if:

$$\forall x{:}\alpha \in \Gamma, y{:}\beta \in \Delta\ .\ x\ \Phi\ y \implies \alpha = \beta.$$

Similarly we define *convertibility of preterms t and u* modulo variable mapping Φ, denoted as $t \overset{\Phi}{\approx} t'$ [4], inductively as:

$$\begin{array}{rl} x \overset{\Phi}{\approx} y, & \text{if } x\ \Phi\ y, \\ f \overset{\Phi}{\approx} f, & \\ @(t_l, t_r) \overset{\Phi}{\approx} @(u_l, u_r), & \text{if } t_l \overset{\Phi}{\approx} u_l \text{ and } t_r \overset{\Phi}{\approx} u_r, \\ \lambda x{:}\alpha.t \overset{\Phi}{\approx} \lambda x{:}\alpha.u, & \text{if } t \overset{\Phi\uparrow^1}{\approx} u. \end{array}$$

with the lifting of variable mapping $\Phi\uparrow^1$ defined as expected.

Now it seems that we can say that two terms $\Gamma \vdash t : \alpha$, $\Delta \vdash u : \beta$ are convertible ($t \overset{\Phi}{\sim} u$) if there exists a variable mapping Φ such that $\Gamma \overset{\Phi}{\approx} \Delta$ and $t \overset{\Phi}{\approx} u$. However we need to be careful. If we require convertibility for all variables

[4] We abuse the notation here and denote preterm convertibility and environment convertibility with the same symbol $\approx$, however depending on the arguments being used it will always be clear which one is to be used.

in domains of environments then the following property, essential for some other results, does not hold:

$$t \overset{\Phi}{\sim} u \ \wedge \ \Phi \subset \Phi' \implies t \overset{\Phi'}{\sim} u.$$

To see that consider terms: $t = x{:}\alpha \vdash c : \alpha$ and $u = x{:}\beta \vdash c : \alpha$ and notice that we have $t \sim_\emptyset u$ but not $t \sim_{\{(x,x)\}} u$ as environments of t and u declare x with different types.

This can be easily repaired if we demand convertibility of environments on free variables only, that is only those declarations that are really used in given term. For a term $\Gamma \vdash t : \alpha$ the environment containing *free variable* of t is denoted as $\mathsf{Vars}(\Gamma \vdash t : \alpha)$ and defined as:

$$\begin{aligned}
\mathsf{Vars}(\Gamma \vdash x : \alpha) &= \{x{:}\alpha\}, \\
\mathsf{Vars}(\Gamma \vdash f : \alpha) &= \emptyset, \\
\mathsf{Vars}(\Gamma \vdash @(t_l, t_r) : \beta) &= \mathsf{Vars}(\Gamma \vdash t_l : \alpha \to \beta) \cdot \mathsf{Vars}(\Gamma \vdash t_r : \alpha), \\
\mathsf{Vars}(\Gamma \vdash \lambda x{:}\alpha.t : \beta) &= \mathsf{Vars}(\Gamma \cdot \{x{:}\alpha\} \vdash t : \alpha \to \beta) \setminus \{x{:}\alpha\}.
\end{aligned}$$

So now we can define *term convertibility*. Two terms $\Gamma \vdash t : \alpha$ and $\Gamma' \vdash t' : \alpha'$ are *convertible* up to a variable mapping Φ, denoted as $\Gamma \vdash t : \alpha \overset{\Phi}{\sim} \Gamma' \vdash t' : \alpha'$ (we will often leave environments and types implicit and write $t \overset{\Phi}{\sim} t'$) iff:

$$\mathsf{Vars}(\Gamma \vdash t : \alpha) \overset{\Phi}{\approx} \mathsf{Vars}(\Gamma' \vdash t' : \alpha') \ \wedge \ t \overset{\Phi}{\approx} t'.$$

Terms $\Gamma \vdash t : \alpha$ and $\Gamma' \vdash t' : \alpha'$ are *convertible* if there exists a variable mapping Φ such that $\Gamma \vdash t : \alpha \overset{\Phi}{\sim} \Gamma' \vdash t' : \alpha'$.

Now we extend the notion of convertibility to substitutions. Substitutions γ and δ are *convertible* with variable mapping Φ, $\gamma \overset{\Phi}{\sim} \delta$, iff:

$$\forall x, y \in \mathcal{V} \ . \ x \ \Phi \ y \implies \begin{cases} x \in Dom(\gamma) \iff y \in Dom(\delta) \\ x/t \in \gamma \wedge y/u \in \delta \implies t \overset{\Phi}{\sim} u \end{cases}.$$

Lemma 6. *Relation $\sim$ is an equivalence relation.*

Proof. For reflexivity we have $t \overset{\Phi}{\sim} t$ with Φ being identity on variables restricted to free variables of t. Symmetry is proven by the observation that if $t \overset{\Phi}{\sim} u$ then $u \overset{\Phi^{-1}}{\sim} t$. For transitivity if $t \overset{\Phi}{\sim} u$ and $u \overset{\Psi}{\sim} w$ then $t \overset{\Phi \cdot \Psi}{\sim} w$. □

Convertibility relation $\sim$ enjoys a number of nice properties including:

- Compatibility with substitution: $t \sim t'$ and $\gamma \sim \gamma'$ implies $t\gamma \sim t'\gamma'$.
- Compatibility with beta-reduction:

$$\left.\begin{array}{c} \Gamma \vdash t : \delta \to_\beta \Delta \vdash u : \eta \\ \Gamma \vdash t : \delta \overset{Q}{\sim} \Gamma' \vdash t' : \delta' \\ \Delta \vdash u : \eta \overset{Q}{\sim} \Delta' \vdash u' : \eta' \\ \Gamma' = \Delta' \end{array}\right\} \implies \Gamma' \vdash t' : \delta' \to_\beta \Delta' \vdash u' : \eta'.$$

– Computability with HORPO, the result analogous to the preceding one just for the HORPO relation, see [13].

The most interesting aspect of formalization of the results from this section is probably the representation of variable mappings in Coq. Variable mappings are partial, injective functions. Moreover we need to be able to compute their inverse for proving symmetry of $\sim$. We know that inverse of any variable mappings exists, as it is an injective function. But this does not make our task any easier as we want to provide a constructive proof and for that we need to be able to compute this inverse: something that clearly cannot be done in full generality. But before giving up constructiveness let us observe that variable mappings operate on environments which are finite. So both domain and codomain of variable mappings are finite and computing inverse of such functions can be accomplished.

To encode variable mappings in Coq we have chosen to look at Φ as a relation. Then computing inverse is trivial as we only need to transpose the relation but we still need to make sure that we can compute $\Phi(x)$ for any x. Let us first present the solution that we employed.

```
Record EnvSubst : Type := build_envSub {
  envSub:     relation nat;
  size:       nat;
  envSub_dec: forall i j, {envSub i j} + {~envSub i j};
  envSub_Lok: forall i j j', envSub i j -> envSub i j' -> j = j';
  envSub_Rok: forall i i' j, envSub i j -> envSub i' j -> i = i';
  sizeOk:     forall i j, envSub i j -> i < size /\ j < size
}.
```

So `envSub` represents Φ function (seen as a relation). Fields `envSub_Lok` and `envSub_Rok` ensure that `envSub` is, respectively, a function and that it is injective. The `size` field is an upper bound on indices of variables both in domain and codomain of `envSub` and `sizeOk` verifies that indeed that is the case. Finally `envSub_dec` states that relation `envSub` is decidable.

Now to compute $\Phi(x)$ we check whether `envSub x y` holds (with the use of `envSub_dec`) for $y \in \{0, \ldots, size - 1\}$. If we find such y then $\Phi(x) = y$ and we know that this y is unique by `envSub_Lok`. On the other hand if no such y exists in this interval then we know that it does not exist at all due to `sizeOk` and we conclude that x is not in the domain of Φ. So using this reasoning we can prove the following lemma:

```
Fact envSubst_dec: forall (i: nat) (Q: EnvSubst),
  {j: nat | envSub Q i j} + {forall j, ~envSub Q i j}.
```

Convertibility of preterms can be expressed in Coq by the following inductive definition. Note the use of lifting of variable mappings in the abstraction case.

```
Inductive conv_term: Preterm -> Preterm -> EnvSubst -> Prop :=
| ConvVar: forall x y S, envSub S x y -> conv_term (%x) (%y) S
```

```
| ConvFun: forall f S, conv_term (^f) (^f) S
| ConvAbs: forall A L R S, conv_term L R (envSubst_lift1 S) ->
    conv_term (\A => L) (\A => R) S
| ConvApp: forall LL LR RL RR S, conv_term LL RL S ->
    conv_term LR RR S -> conv_term (LL @@ LR) (RL @@ RR) S.
```

For convertibility of environments we need only to look at free variables of terms (`activeEnv` corresponding to Vars; see [13] for the Coq definition) and not on full environments.

```
Definition activeEnv_compSubst_on M N x y :=
  forall A, activeEnv M |= x := A <-> activeEnv N |= y := A.
Definition conv_env (M N: Term) (S: EnvSubst) : Prop :=
  forall x y, envSub S x y -> activeEnv_compSubst_on M N x y.
```

Finally convertibility of terms demands that both preterms and environments are convertible.

```
Definition terms_conv_with S M N :=
  conv_term (term M) (term N) S /\ conv_env M N S.
Definition terms_conv M N := exists S, terms_conv_with S M N.
```

Then `terms_conv` is proven to be an equivalence relation (actual Coq proof is somehow more complicated than what Lemma 6 would suggest) and it is registered as a setoid. In Coq working with structures for which Leibniz equality does not denote the intended equality is not very easy. Setoid is an extension that makes it easier by allowing to register an equivalence relation along with some functions compatible with it (morphisms). Then one can replace a term by an equivalent one in arguments of such functions as easily as if they were equal.

6 Conclusions

We presented an axiom-free, fully constructive Coq formalization of $\lambda^{\rightarrow}$ being the result of the author's efforts during past two years. Results such as a typed substitution, convertibility of terms extending the notion of α-convertibility to free variables, a proof of strong normalization of $\lambda^{\rightarrow}$ and many more has been formalized resulting in a rather comprehensive formalization of simply typed lambda calculus.

The main question that arises is whether the approach taken for this formalization has been an appropriate one. For that the crucial point is the representation of terms. We are convinced that de Bruijn indices were a better choice than named variables. Dealing with lifting of terms required some effort but it was clearly much less involving than dealing explicitly with α-conversion would have been (especially in the context of the higher-order rewriting where this formalization was used). Another approach to representing terms with binders is higher-order abstract syntax (HOAS) [15]. It is not immediately clear whether this approach could have been successfully employed in our formalization.

The main difficulty throughout the formalization was the need of constant reasoning with dependent types, that quickly becomes difficult in Coq. This is however due to the nature and complexity of the formalized theory and cannot be helped. However we believe that the reproduction of those result, with the advantage of all the lessons learned, would take substantially less time.

On the one hand this development has already been successfully used in the formalization of the higher-order recursive path ordering [14], which is the contribution to the CoLoR project. On the other hand it is fully autonomous and after submission as Coq contribution can be reused in further developments dealing with $\lambda^{\rightarrow}$.

References

1. CoLoR: a Coq library on rewriting and termination. `http://color.loria.fr`.
2. T. Altenkirch. A formalization of the strong normalization proof for system f in lego. In *TLCA*, volume 664 of *LNCS*, pages 13–28, 1993.
3. T. Altenkirch. Proving strong normalization of cc by modifying realizability semantics. In *TYPES*, volume 806 of *LNCS*, pages 3–18, 1993.
4. T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS*, pages 303–310, 2001.
5. H. P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science (vol. II)*, pages 117–309, 1992.
6. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Phd thesis, Université Paris 7, November 1999.
7. U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82:25–49, 2006.
8. N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
9. A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.
10. Brian E. Aydemir et. al. Mechanized metatheory for the masses: The PoplMark challenge. In *TPHOLs*, pages 50–65, 2005.
11. J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *LICS*, pages 402–411, 1999.
12. J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. 2005. Submitted, `http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/biblio.html`.
13. A. Koprowski. Certified higher-order recursive path ordering. Technical report, Eindhoven University of Technology, Eindhoven, The Netherlands, May 2006. To appear, `http://www.win.tue.nl/~akoprows/coq-horpo.pdf`.
14. A. Koprowski. Certified higher-order recursive path ordering. In *RTA*, LNCS, 2006. To appear.
15. F. Pfenning and C. Elliot. Higher-order abstract syntax. In *PLDI '88*, pages 199–208, 1988.
16. K. Stoevring, O. Danvy, and M. Biernacka. Program extraction from proofs of weak head normalization. In *MFPS*, ENTCS, 2005.
17. The Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.0*. Available at http://coq.inria.fr.