Coq formalization of the higher-order recursive path ordering

Coq Formalization of the Higher-Order Recursive Path Ordering

Adam Koprowski

Abstract

Recursive path ordering (RPO) is a well-known reduction ordering introduced by Dershowitz [14], that is useful for proving termination of term rewriting systems (TRSs). Jouannaud and Rubio generalized this ordering to the higher-order case thus creating the higher-order recursive path ordering (HORPO) [19]. They proved that this ordering can be used for proving termination of higher-order TRSs which essentially comes down to proving well-foundedness of the union of HORPO and the β-reduction relation of the simply typed lambda calculus ($\lambda^{\rightarrow}$). This result entails well-foundedness of RPO and termination of $\lambda^{\rightarrow}$.

This paper describes author's undertaking of providing a complete, axiom-free, fully constructive formalization of those results in the theorem prover Coq. Formalization is complete and hence it contains all the dependant results. It can be divided into three parts:

- finite multisets and two variants of multiset extensions of a relation,
- $\lambda^{\rightarrow}$ with termination of β as the main result,
- HORPO with proof of well-foundedness of its union with β-reduction. Also decidability of HORPO has been proven and due to the constructive nature of this proof a certified algorithm to verify whether two terms can be oriented with HORPO can be extracted from this proof.

Contents

Chapter 1

Introduction

The recursive path ordering (RPO) goes back to Dershowitz [14]. It is a well-known reduction ordering for first-order term rewriting systems (TRSs). Jouannaud and Rubio generalized this ordering to the higher-order case thus creating the higher-order recursive path ordering (HORPO) [19, 20].

This paper describes the formalization of this ordering along with the proof of its well-foundedness. In our attempt to formalize those results we were interested in the first, simplest definition presented in [19] without all the improvements introduced in the same publication and later on in [20].

The formalization has been carried out in the theorem prover Coq [15], which is based on the calculus of inductive constructions [10]. This intuitionistic type theory allows for use of dependent types and is constructive hence all the proofs in the formalization are constructive. Version 8.0 of the prover has been used.

This report contains explanation of both the theory and its formalization. The author tried to separate the two. The details of the formalization succeed the explanation of the theory at the end of (almost) every section and both parts are separated by a horizontal line labelled "Coq" such as this:

Coq

Hence the reader interested only in the theoretical part can safely skip the explanation of the formalization.

1.1 Motivation

One may ask what is the motivation behind such a formalization effort. We would like to identify three main motivational factors for this work.

- **Advancement of formal methods.**
  As with everything else formal methods in general and theorem proving in particular need some stimulus for their growth. Theorem proving is still a rather laborious task but with constant improvement of technology a future when the critical systems will be proven correct before employing is not impossible. And the progress in the area of proof assistance technology is triggered by big developments accomplished with existing theorem provers.

- **Verification of the proof.**
  Especially for complicated proofs that are not very well known (and thus thoroughly checked by the community), such as the work in [19], the goal may simply be to verify the correctness of the paper proof. The results from [19, 20] are impressive and complicated and as such are

inevitably subject to some small slips. This justifies the effort of verification of such results. Indeed in the course of formalization we were able to detect a small flaw, concerning the use of multiset extension of an arbitrary relation, that could be easily repaired (we will discuss it shortly in Sections 3.2.1 and 7.3). In general [19, 20] turned out to be a very favorable subject for formalization and the structure of the proofs could be followed to the letter in the formalization process.[1]

- **Contribution to the CoLoR [1] project.**
  CoLoR is a project aiming at proving theoretical results from the area of term rewriting in the theorem prover Coq. The ultimate goal is to (automatically) transform termination proof candidates produced by termination tools into formal Coq proofs certifying termination. This requires formalization of the term rewriting theory and this development is a contribution to the CoLoR library.

1.2 Overview

The development can be divided intro four main parts:

- **Auxiliary results** (generally not discussed in this paper)
  - Lexicographic order (Chapter 2).
  - Many operations and properties concerning lists and relations that were not present in the Coq standard library.
- **Multisets and multiset extensions of a relation** (Chapter 3).
  - Multisets as an abstract data-type (Section 3.1).
  - Concrete implementation of multisets using lists.
  - Definition of two variants of multiset extension of a relation (Section 3.2.1).
  - Multiset extensions preserve orders (Section 3.2.3).
  - Multiset extensions preserve well-foundedness (Section 3.3).
- **Simply typed lambda calculus** (Chapter 4).
  - Definition of simply typed lambda terms over an arbitrary signature with constants and typing a'la Church (Section 4.1).
  - Properties of environments, subterm relation and many further definitions and results (Section 4.2).
  - Typing properties: uniqueness of types, decidability of typing (Section 4.2.2).
  - Many-variable, typed substitution (Section 4.3).
  - Convertibility relation on typed terms extending the concept α-convertibility to free variables (Section 4.4).
  - β-reduction and its properties (Section 4.5).
- HORPO (Chapters 5-7).
  - Terms with arity encoded by simply typed lambda terms (Section 5.1).
  - Introduction to the higher-order rewriting framework of algebraic functional systems (AFSs) by Jouannaud and Okada [18] (Section 5.2). Note that this is not part of the formalization and is included in this paper only to provide the context for the definition of HORPO.

[1] Obviously providing formal proofs requires to be more explicit and to include all the results that in normal presentation would be omitted as considered to be straightforward or irrelevant.

– Computability predicate proof method by Tait and Girard and some computability properties used in the proof of well-foundedness of the union of HORPO and β-reduction (Chapter 6).
– Definition of the HORPO ordering along the lines of [19] (Section 7.1).
– Proofs of some properties of HORPO including its decidability (Section 7.2).
– Proof of well-foundedness of the union of HORPO and β-reduction (Section 7.3).

Coq

Figure 1.1 depicts[2] relative sizes of those four parts measured by the size of Coq scripts (with one box representing one file and the area of the box proportional to the file size). Precise figures are given in the table below:

Part of the development	No. of files	Lines of code	Total files size
Auxiliaries	6	2,780	70,455
Multisets	6	4,432	130,554
Simply typed lambda calculus	17	13,951	440,295
HORPO	4	3,027	100,239
TOTAL	33	24,190	741,543

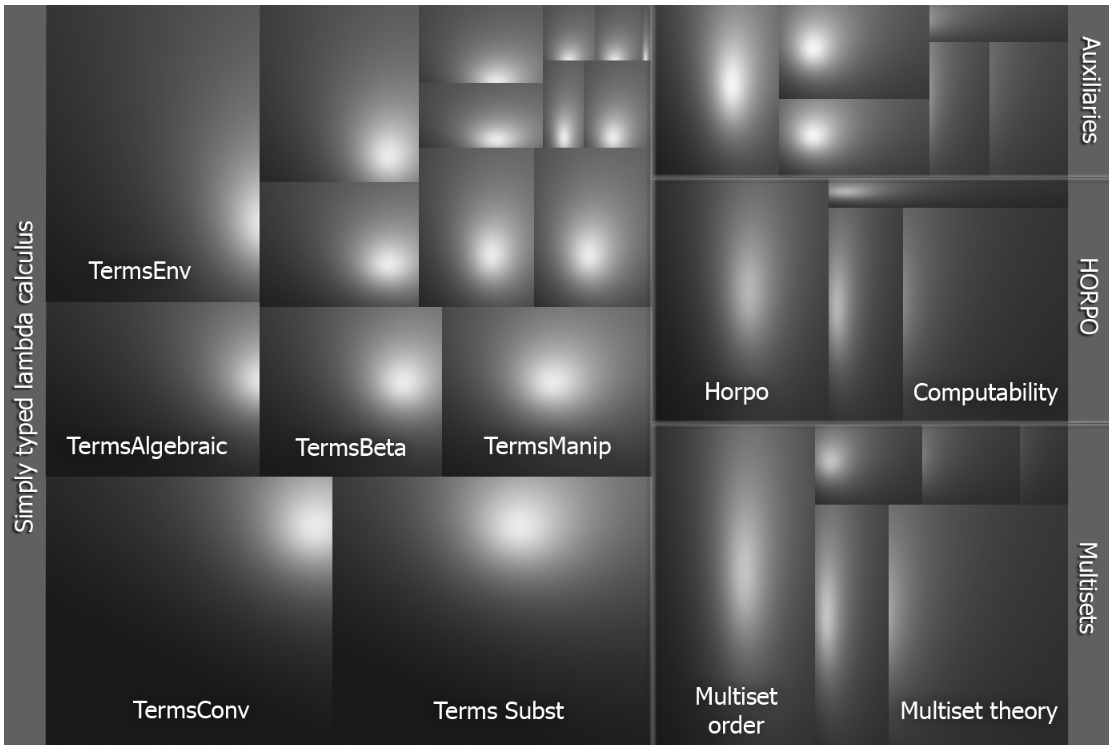

Figure 1.1: Size of the Coq development.

1.3 Related Work

Simply typed lambda calculus and (first order) RPO has been subject to many formalization efforts to date. However to the best of our knowledge our contribution is the first formalization of the

[2]The visualization has been accomplished using SequoiaView program developed at the Eindhoven University of Technology.

higher order variant of recursive path order. Below we list few somehow related formalizations. We begin with listing some formalizations of typed lambda calculi.

- Berger et al. in their recent work [8] proved strong normalization of $\lambda^{\rightarrow}$ in three different theorem provers, including Coq and from those proofs machine-extracted implementations of normalization algorithms. Their formalization is closely related to our formalization of $\lambda^{\rightarrow}$. 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 part of the formalization presented in this paper concerning $\lambda^{\rightarrow}$ 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 in mind.
- Another source of formalizations of lambda calculi is the PoplMark challenge [16]: a set of benchmarks for measuring progress in the area of formalizing metatheory of programming languages. Among numerous submissions to PoplMark there are even few using Coq and de Bruijn representation of terms. The comparison however is difficult: although the benchmark is designed for a richer type system (System $F_{<:}$) it focuses on completely different aspects.
- Other formalizations include strong normalization proofs for calculi like: Calculus of Constructions [7], [3]; System F [2]; typed λ-calculus with co-products [4] and λ-calculi with weak reduction strategies [33].

There are also some formalizations of RPO that are worth mentioning here.

- Murthy [26] formalizes a classical proof of Higman's lemma, a specific instance of Kruskal's tree theorem, in a classical extension of Nuprl 3. The classical proof is due to Nash-Williams and uses a minimal bad sequence argument. The formalized classical proof was automatically translated into a constructive proof using Friedman's A-translation.
- Berghofer [9] presents a constructive proof of Higman's lemma in Isabelle. The constructive proof is due to Coquand and Fridlender.
- Persson [29] presents a constructive proof of well-foundedness of a general form of recursive path relations. This proof is very similar to, and independently obtained, of the specialization to the first-order case of the proof of well-foundedness of the HORPO by Jouannaud and Rubio [19]. The proof in [29] is extracted from the classical proof using a minimal bad sequence argument by using open induction due to Raoult [32]. Persson presents an abstract formalization of well-foundedness of recursive path relations in the proof-checker Agda.
- Leclerc [25] presents a formalization of well-foundedness of the multiset path ordering in Coq. The focus is on giving upper bounds for descending sequences.
- De Kleijn in her Master Thesis [22] shows well-foundedness of RPO in Coq. Her development however is not complete and contains a number of axioms.
- Coupet-Grimal and Delobel [13] have provided a full development of well-foundedness of RPO in Coq. In their formalization they use multisets and a multiset extension of a relation developed by the author and described in Chapter 3 of this paper.

Chapter 2

Preliminaries

We begin with the definition of the lexicographic order on the product of two sets.

Definition 2.1.
Let A and B be two sets equipped with strict orders: $>_A$ and $>_B$ respectively. Then the lexicographic order $>_{lex}$ on pairs $A \times B$ is defined as:

$$(a, b) >_{lex} (a', b') \iff a >_A a' \lor (a = a' \land b >_B b')$$

An important property of lexicographic order is that it preserves well-foundedness.

Theorem 2.1.
Let A and B be two sets equipped with strict orders: $>_A$ and $>_B$ respectively. If $>_A$ and $>_B$ are well-founded then $>_{lex}$ is well-founded.

Now we will introduce the notion of well-founded induction. Let W be a set and $>$ a relation on W. Define $<$ as a transposition of $>$. Now we define a well-founded part of W with respect to $<$, denoted as $\mathcal{W}^W_<$ inductively as follows:

$$\frac{\forall y < x \ . \ y \in \mathcal{W}^W_<}{x \in \mathcal{W}^W_<}$$

Clearly $<$ is well-founded if and only if $\mathcal{W}^W_< = W$.

Now the following induction principle of well-founded part induction will be crucial for the proof of well-foundedness of multiset extensions in Section 3.3.

$$\frac{\forall x \in \mathcal{W}^W_< \ . \ (\forall y < x \ . \ P(y)) \implies P(x)}{\forall x \in \mathcal{W}^W_< \ . \ P(x)}$$

Coq

The notion of well-foundedness in the above sense is present in the standard library of `Coq` in the module `Coq.Init.Wf`. The membership in $\mathcal{W}^W_<$ is expressed by the following accessibility predicate:

```
Variable A : Set.
Variable R : A -> A -> Prop.
Inductive Acc : A -> Prop :=
   Acc_intro : forall x:A, (forall y:A, R y x -> Acc y) -> Acc x.
```

Well-foundedness again means that all elements are accessible:

```
Definition well_founded := forall a:A, Acc a.
```

Finally the induction principle generated by Coq for the definition of Acc corresponds to well-founded part induction introduced earlier.

```
Acc_ind: forall (A: Set) (R: A -> A -> Prop) (P: A -> Prop),
  (forall x: A,
    (forall y: A, R y x -> Acc R y) ->
    (forall y: A, R y x -> P y) ->
    P x
  ) ->
  forall a: A, Acc R a -> P A.
```

Chapter 3

Multisets and Multiset Extensions of a Relation

3.1 Multisets

Multisets extend the notion of a set by relaxing the condition that all elements are pairwise different. So in a multiset every element may occur a number of times and the number of its occurrences will be called a *multiplicity* of the element. Formally a multiset over a given domain A is represented by a function assigning a natural number (multiplicity) to every element of the domain. This function is a multiset counterpart of a characteristic function for sets. In the following we fix a set A.

Definition 3.1 (Multisets)**.**
A multiset *M over a domain set A is a function $M : A \to \mathbb{N}$.*

A finite multiset *is a multiset for which there are only finitely many x such that $M(x) > 0$. We denote the set of finite multisets over A by $\mathbb{M}_A$.*

In this paper we focus on finite multisets only. Although some theory about multisets extends to infinite multisets, the parts we are interested in do not: only finite multisets can be treated as a data-type and the crucial property of a multiset extension of a relation, preservation of well-foundedness, does not hold if we allow multisets containing infinitely many elements.

Coq

We continue with showing how multisets can be defined in Coq. We use Coq module mechanism to develop an abstract specification of multisets. We create a module type parameterized by a set A equipped with decidable equality. This module declares a `Multiset` data-type and the crucial function, `mult`, which given an element and a multiset returns the multiplicity of a given element in that multiset:

```
Parameter Multiset : Set.
Parameter mult : A -> Multiset -> nat.
```

Further the specification calls for an existence of an empty multiset and of the following operations on multisets: equality, construction of a singleton multiset and union, intersection and difference of multisets. The summary of those operations with corresponding Coq declarations is presented in Table 3.1. The specification of those operations is given in terms of the `mult` function and is presented in Table 3.2.

Note that those operations are not completely independent and for instance intersection can be defined using the difference operator as:

$$M \cap N := M \setminus (M \setminus N)$$

Table 3.1: Primitive operations on multisets.

Operation	Coq declaration	Coq notation
$M(x)$	`mult: A -> Multiset -> nat`	`x/M`
$M = N$	`meq: Multiset -> Multiset -> Prop`	`M =mul= N`
$\emptyset$	`empty: Multiset`	`empty`
$\{\{x\}\}$	`singleton: A -> Multiset`	`{{ x }}`
$M \cup N$	`union: Multiset -> Multiset -> Multiset`	`M + N`
$M \cap N$	`intersection: Multiset -> Multiset -> Multiset`	`M # N`
$M \setminus N$	`diff: Multiset -> Multiset -> Multiset`	`M - N`

Both operations have been kept for efficiency reasons[1] but the above definition has been proven to realize the specification of the intersection operation.

One more additional requirement is the validity of the following inductive reasoning principle for multisets (P being an arbitrary proposition over multisets).

$$\frac{P\ \emptyset \qquad \forall M \in \mathbb{M}_A\ .\ \forall a \in A\ .\ P\ M \implies P\ (M \cup \{\{a\}\})}{\forall M \in \mathbb{M}_A\ .\ P\ M}$$

So if we can prove that a property holds for an empty multiset and if assuming it holds for M we can prove that it also holds for $M \cup \{\{a\}\}$ (for an arbitrary a) then it holds for every multiset. This induction principle effectively restricts multisets to being finite. It is stated in Coq as follows:

```
Axiom mset_ind_type: forall P : Multiset -> Type,
  P empty ->
  (forall M a, P M -> P (M + {{a}})) ->
  forall M, P M.
```

Using those primitives some additional operations such as: multiset membership, insert and remove functions, two element multiset constructor, conversion from and to lists and cardinality are defined. They are summarized in Table 3.3.

Then a simple list implementation of multisets has been provided. Multisets are represented as lists, the equality of multisets corresponds to the permutation predicate on lists, the union operation is realized by list concatenation and all other operations are implemented in a fairly straightforward way.

Also a number of simple properties about multisets has been proven. They depend only on the axiomatic specification. This ensures that given another implementation of multisets, say, more efficient one, those results carry over automatically. The same holds for all the rest of the multiset theory presented in the following sections so all the results are really independent of the actual implementation of the multiset data-type.

[1]It may be possible to provide a more efficient implementation for computing intersection of two multisets than by the above computation.

Table 3.2: Specification of primitive operations on multisets.

Operation	Specification	Coq specification

$M = N$	$M = N \iff \forall x.\ M(x) = N(x)$	`Axiom mult_eqA_compat:`
		`   x =A= y ->`
		`   x/M = y/M.`
		`Axiom meq_multeq:`
		`   M =mul= N ->`
		`  (forall x, x/M = x/N).`

$\emptyset$	$\emptyset(x) := 0$	`Axiom empty_mult:`
		`  x/empty = 0.`

$\{\{x\}\}$	$\{\{x\}\}(x) := 1$
	$\{\{x\}\}(y) := 0 \quad x \neq y$
		`Axiom singleton_mult_in:`
		`  x =A= y -> x/{{y}} = 1.`
		`Axiom singleton_mult_notin:`
		`  ~x =A= y -> x/{{y}} = 0.`

$M \cup N$	$(M \cup N)(x) := M(x) + N(x)$	`Axiom union_mult:`
		`  x/(M+N) = (x/M + x/N)%nat.`

$M \cap N$	$(M \cap N)(x) := min\{M(x), N(x)\}$	`Axiom intersection_mult:`
		`  x/(M#N) = min (x/M) (x/N).`

$M \setminus N$	$(M \setminus N)(x) := M(x) \ominus N(x)$
	where: $m \ominus n := \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{otherwise} \end{cases}$
		`Axiom diff_mult:`
		`  x/(M-N) = (x/M - x/N)%nat.`

3.2 Multiset Extensions of a Relation

3.2.1 Definition

In this section we present two rather standard definitions (see for instance [5]) of an extension of a relation on elements to a relation on multisets of elements. It is usual to define those extensions for orderings, however we define them for arbitrary relations and only in Section 3.2.3 we study some properties valid only in case the underlying relation is an order.

The intuition behind the subsequent definition is as follows: to prove that a multiset M is bigger than N we are allowed to remove an arbitrary element a from M and any number of elements from N that are smaller than a. If by repetition of this process we can make those two multisets equal (in particular: empty) then we have proven that M is bigger than N. The single step of this process is captured in the following definition by the notion of the *multiset reduction* and since we may use more than one step the actual *multiset extension of a relation* is just a transitive closure of that multiset reduction.

Definition 3.2 (Multiset extension of a relation, $>_{mul}$).
Let $>$ be a relation on A. We define multiset reduction *relation $\rhd_{mul}$ on $\mathbb{M}_A$ as:*

$$M \rhd_{mul} N \text{ iff } \begin{cases} \exists X, Y \in \mathbb{M}_A; a \in A \text{ . such that:} \\ -)\ \ M = X \cup \{\{a\}\} \\ -)\ \ N = X \cup Y \\ -)\ \ \forall y \in Y \text{ . } a > y \end{cases}$$

We will say that the triple $\langle X, a, Y\rangle_{mul}$ proves $X \cup \{\{a\}\} = M \rhd_{mul} N = X \cup Y$.

Now multiset extension of a relation *($>_{mul}$) is the transitive closure of the multiset reduction, so $>_{mul} = \rhd^+_{mul}$.*

Table 3.3: Derived operations on multisets.

Description	Operation	Coq declaration
Membership	$a \in M$	`member: A -> Multiset -> Prop`
Element insertion	$M \cup \{\{a\}\}$	`insert: A -> Multiset -> Multiset`
Element removal	$M \setminus \{\{a\}\}$	`insert: A -> Multiset -> Multiset`
Pair constructor	$\{\{a, b\}\}$	`pair: A -> A -> Multiset`
Conversion from list		`list2multiset: list A -> Multiset`
Conversion to list		`multiset2list: Multiset -> list A`
Multiset cardinality		`card: Multiset -> int`

Another way of defining a multiset extension is to combine together the "small steps" from the above definition and obtain a "big steps" variant as presented below.

Definition 3.3 (Multiset extension of a relation, $>_{MUL}$).
Let $>$ be a relation on A. We define the multiset extension of a relation $>_{MUL}$ as a relation on $\mathbb{M}_A$:

$$M >_{MUL} N \textit{ iff } \begin{cases} \exists X, Y, Z \in \mathbb{M}_A \text{ . such that:} \\ -)\ Y \neq \emptyset \\ -)\ M = X \cup Y \\ -)\ N = X \cup Z \\ -)\ \forall z \in Z \text{ . } \exists y \in Y \text{ . } y > z \end{cases}$$

We will say that the triple $\langle X, Y, Z\rangle_{MUL}$ proves $X \cup Y = M >_{MUL} N = X \cup Z$.

First let us remark that for $>$ being an arbitrary relation $>_{mul}$ and $>_{MUL}$ do differ. Take $A = \{a, b, c\}$ and $> = \{(a, b), (b, c)\}$, then $\{a\} >_{mul} \{c\}$ because $\{a\} \rhd_{mul} \{b\}$ and $\{b\} \rhd_{mul} \{c\}$, but not $\{a\} >_{MUL} \{c\}$. If $>$ is transitive (particularly, if it is an order) then those definitions coincide as we will see in Section 3.2.3.

Coq

We will show how definitions of $>_{mul}$ and $>_{MUL}$ can be expressed in Coq. First we assume an arbitrary relation on A, $>$.

```
Variable gtA : A -> A -> Prop.
Notation "X >A Y" := (gtA X Y) (at level 50).
```

Then the definition of $>_{MUL}$ can be expressed as follows:

```
Inductive MultisetRedGt (M N: Multiset) : Prop :=
| MSetRed: forall X a Y,
    M =mul= X + {{a}} ->
    N =mul= X + Y ->
    (forall y, y in Y -> a >A y) ->
    MultisetRedGt M N.

Definition MultisetGt := clos_trans Multiset MultisetRedGt.
Notation "X >mul Y" := (MultisetGt X Y) (at level 70) : mord_scope.
```

Thanks to the use of Coq notations (=mul= for multiset equality, + for multiset union, {{_}} for singleton and _ in _ for multiset membership; see Table 3.1) the fact that MultisetRedGt is a Coq coding of $\rhd_{mul}$ and MultisetGt of $>_{mul}$ is very easy to recognize (clos_trans is a definition of transitive closure of a relation from the Coq standard library).

Similarly we have a straightforward representation of $>_{MUL}$:

```
Inductive MultisetGT (M N: Multiset) : Prop :=
| MSetGT: forall X Y Z,
   Y <>mul empty ->
   M =mul= X + Y ->
   N =mul= X + Z ->
   (forall z, z in Z -> (exists2 y, y in Y & y >A z)) ->
   MultisetGT M N.
Notation "X >MUL Y" := (MultisetGT X Y) (at level 70) : mord_scope.
```

3.2.2 Properties of Multiset Extensions of a Relation

As remarked in the previous section $>_{mul}$ and $>_{MUL}$ differ for non-transitive relations. For now we will prove that $>_{MUL}$ is a subset of $>_{mul}$. We will use this result in Section 3.2.3 to prove equivalence of $>_{mul}$ and $>_{MUL}$ for orders and also to conclude well-foundedness of $>_{MUL}$ from well-foundedness of $>_{mul}$ in Section 3.3. We begin with an auxiliary result.

Lemma 3.1.
Let $X, Y \in \mathbb{M}_A$ and $a \in A$. If $\forall y \in Y \,.\, \exists x \in (X \cup \{\{a\}\}) \,.\, x > y$ then there exist multisets Y_X and Y_a such that:

- $Y = Y_a \cup Y_X$,
- $\forall y \in Y_a \,.\, a > y$,
- $\forall y \in Y_X \,.\, \exists x \in X \,.\, x > y$.

Proof. Induction on Y.

- If $Y = \emptyset$ then simply take $Y_a = Y_X = \emptyset$.
- If $Y = Y' \cup \{\{y\}\}$ then by the induction hypothesis we get Y'_a and Y'_X such that $Y' = Y'_a \cup Y'_X$ and the last two conditions of the lemma are satisfied. By assumption we get that $\exists x \in (X \cup \{\{a\}\}) \,.\, x > y$ since $y \in Y$. We have two cases:
  - If $x \in X$ then take $Y_a = Y'_a$ and $Y_X = Y'_X \cup \{\{y\}\}$.
  - If $x = a$ then take $Y_a = Y'_a \cup \{\{y\}\}$ and $Y_X = Y'_X$.

□

Lemma 3.2.
$\forall M, N \in \mathbb{M}_A \,.\, M >_{MUL} N \implies M >_{mul} N$.

Proof. We have $M = X \cup Y >_{MUL} X \cup Z = N$ with $\langle X, Y, Z\rangle_{MUL}$. We prove $X \cup Y >_{mul} X \cup Z$ by induction on the multiset Y. Base case is easily discarded as $Y \neq \emptyset$ by the definition of $>_{MUL}$.

For the induction step we have $X \cup Y \cup \{\{a\}\} >_{MUL} X \cup Z$ by $\langle X, Y \cup \{\{a\}\}, Z\rangle_{MUL}$ and we need to show $X \cup Y \cup \{\{a\}\} >_{mul} X \cup Z$. We distinguish two cases:

- $Y = \emptyset$. Then triple $\langle X, a, Z\rangle_{mul}$ proves $X \cup Y \cup \{\{a\}\} >_{mul} X \cup Z$.

- $Y \neq \emptyset$. By Lemma 3.1 we split Z into Z_a and Z_Y such that $Z = Z_a \cup Z_Y$, $\forall z \in Z_a \,.\, a > z$ and $\forall z \in Z_Y \,.\, \exists y \in Y \,.\, y > z$. Now we will continue with showing that $X \cup Y \cup \{\{a\}\} >_{mul} X \cup Z_Y \cup \{\{a\}\} >_{mul} X \cup Z$ which by transitivity of $>_{mul}$ will complete the proof.
  - $X \cup Y \cup \{\{a\}\} >_{mul} X \cup Z_Y \cup \{\{a\}\}$ by application of the induction hypothesis. For that we need to show $X \cup Y \cup \{\{a\}\} >_{MUL} X \cup Z_Y \cup \{\{a\}\}$ which is proven by $\langle X \cup \{\{a\}\}, Y, Z_Y\rangle_{MUL}$ as both $Y \neq \emptyset$ and $\forall z \in Z_Y \,.\, \exists y \in Y \,.\, y > z$ by assumption.
  - $X \cup Z_Y \cup \{\{a\}\} >_{mul} X \cup Z$ is easily proven by $\langle X \cup Z_Y, a, Z_a\rangle_{mul}$.

□

Clearly $>_{MUL}$ is decidable provided we can decide $>$ as we will show in the following Theorem.

Theorem 3.3.
If $>$ is a decidable relation then $>_{MUL}$ is also decidable.

Proof. To decide whether $M >_{MUL} N$ we need to search for a witness $\langle X, Y, Z\rangle_{MUL}$. Since $X, Y \subseteq M$ and $X, Z \subseteq N$ there are only finitely many potential witnesses and we can consider all of them.[2] □

Let us conclude this section with a simple property of $>_{mul}$ and $>_{MUL}$ stating that every element in a smaller multiset is a reduct of some element in a bigger multiset.

Proposition 3.4. *(i) If $M >_{mul} N$ then $\forall n \in N \,.\, \exists m \in M \,.\, m >^* n$.*

(ii) If $M >_{MUL} N$ then $\forall n \in N \,.\, \exists m \in M \,.\, m \geq n$.

Proof. (ii) is immediate from the definition of $>_{MUL}$. For (i) first let us note that if $M \rhd_{mul} N$ then $\forall n \in N \,.\, \exists m \in M \,.\, m \geq n$. The main goal easily follows as $>_{mul} = \rhd^*_{mul}$. □

Note that the conclusion in case (i) of the above lemma being $m >^* n$ and not $m \geq n$ is essential and for non-transitive $>$ makes a difference. This property in this wrong form was (implicitly) stated in [19, (page 407,case 3 in the proof of Property 3.5)]. Although there it is only a very minor flaw that can be very easily repaired, it shows that one need to be careful with reasoning about multiset extensions of non-transitive relations.

Coq

We would like to make a short comment on the proof of decidability of the multiset extension $>_{MUL}$ (assuming decidability of $>$).

```
Variable gtA_dec : forall (a b: A), {a >A b} + {a <=A b}.
Lemma mOrd_dec : forall M N, {M >MUL N} + {~M >MUL N}.
```

Since the proof is constructive it provides a decision procedure for the problem: "given two multisets M and N does $M >_{MUL} N$ hold?". The algorithm essentially considers all possible ways of splitting M into X and Y such that $X \subseteq N$ and Y is not empty and then for every element $z \in Z = N \setminus X$ looks for $y \in Y$ such that $y > z$ by examining all elements in Y. Note that there is a much more efficient procedure in case $>$ is linear.

[2]The Coq proof (and as a consequence – an algorithm) is slightly more involved. See notes on the Coq implementation below.

3.2.3 Multiset Extensions of Orders

In this section we investigate properties of multiset extensions in case $>$ is an order. We will prove that then multiset extensions are also orders and therefore in that context are often called *multiset orders*. We will also show that $>_{mul}$ and $>_{MUL}$ coincide if $>$ is transitive (so in particular if it is an order).

We begin by proving transitivity of $>_{MUL}$. For that we first introduce an auxiliary lemma.

Lemma 3.5.
$\forall L, R, U, D \in \mathbb{M}_A \ . \ L \cup R = U \cup D \implies R = (R \cap D) \cup (U \setminus L)$.

Proof. To prove $R = (R \cap D) \cup (U \setminus L)$ we will prove $R(x) = (R(x) \cap D(x)) \cup (U(x) \setminus L(x))$ for an arbitrary x. Using definitions of basic multiset operations we get $(R(x) \cap D(x)) \cup (U(x) \setminus L(x)) = \min\{R(x), D(x)\} + (U(x) \ominus$ [3]$L(x))$. We proceed by case analysis. Note that $L(x) + R(x) = U(x) + D(x)$, which holds by assumption, allows us to discard two cases which contradict this equality.

- $U(x) \geq L(x), R(x) \geq D(x)$.

$$\begin{aligned}\min\{R(x), D(x)\} + (U(x) \ominus L(x)) &= \min\{R(x), D(x)\} + (U(x) - L(x)) \\ &= D(x) + U(x) - L(x) \\ &= L(x) + R(x) - L(x) \\ &= R(x)\end{aligned}$$

- $U(x) < L(x), R(x) < D(x)$. Trivial.

□

Lemma 3.6.
Let $>$ be a transitive relation. Then $>_{MUL}$ is also transitive.

Proof. We have $M >_{MUL} N >_{MUL} P$ with $\langle X_1, Y_1, Z_1\rangle_{MUL}$ proving $M >_{MUL} N$ and $\langle X_2, Y_2, Z_2\rangle_{MUL}$ proving $N >_{MUL} P$. We claim that then $\langle X_1 \cap X_2, Y_1 \cup (Y_2 \setminus Z_1), Z_2 \cup (Z_1 \setminus Y_2)\rangle_{MUL}$ proves $M >_{MUL} P$ as:

- $Y_1 \cup (Y_2 \setminus Z_1) \neq \emptyset$ as $Y_1 \neq \emptyset$.
- $M = X_1 \cup Y_1 = (X_1 \cap X_2) \cup (Y_2 \setminus Z_1) \cup Y_1$. First equality is due to $M >_{MUL} N$ with $\langle X_1, Y_1, Z_1\rangle_{MUL}$. The second one is by Lemma 3.5 as $X_1 \cup Z_1 = N = X_2 \cup Y_2$.
- $N = X_1 \cup Z_1 = (X_1 \cap X_2) \cup (Z_1 \setminus Y_2) \cup Z_2$. Analogously to the above case.

□

Now we can prove equivalence of $>_{mul}$ and $>_{MUL}$ for transitive relations.

Theorem 3.7.
Let $>$ be transitive then $\forall M, N \in \mathbb{M}_A \ . \ M >_{mul} N \iff M >_{MUL} N$.

Proof. ($\Rightarrow$) Induction on $M >_{mul} N$. Either $M \rhd_{mul} N$ with $\langle X, a, Y\rangle_{mul}$ but then $M >_{MUL} N$ with $\langle X, \{\{a\}\}, Y\rangle_{MUL}$ or $M >_{mul} M' >_{mul} N$ but then $M >_{MUL} M' >_{MUL} N$ by the induction hypothesis and we conclude by transitivity of $>_{MUL}$, Lemma 3.6 (note the use of transitivity of $>$ here).

($\Leftarrow$) Lemma 3.2. □

[3]See the specification of the difference operator in Figure 3.2 for the definition of $\ominus$.

Now we continue with showing that if $>$ is a strict order then so are $>_{MUL}$ and $>_{mul}$. We need an auxiliary lemma first.

Lemma 3.8.

$$\forall M, N, P \in \mathbb{M}_A \ . \ M >_{MUL} N \iff M \cup P >_{MUL} N \cup P$$

Proof. By induction on P. Essentially we need to prove:

$$M >_{MUL} N \iff M \cup \{\{a\}\} >_{MUL} N \cup \{\{a\}\}$$

($\Rightarrow$) We have $\langle X, Y, Z\rangle_{MUL}$ proving $M >_{MUL} N$ but then $\langle X \cup \{\{a\}\}, Y, Z\rangle_{MUL}$ proves $M \cup \{\{a\}\} >_{MUL} N \cup \{\{a\}\}$.

($\Leftarrow$) We have $\langle X, Y, Z\rangle_{MUL}$ proving $M \cup \{\{a\}\} >_{MUL} N \cup \{\{a\}\}$. If $a \in X$ then clearly $\langle X \setminus \{\{a\}\}, Y, Z\rangle_{MUL}$ proves $M >_{MUL} N$. If $a \notin X$ then from the definition of $>_{MUL}$, $a \in Y$ and $a \in Z$ and $\langle X, Y \setminus \{\{a\}\}, Z \setminus \{\{a\}\}\rangle_{MUL}$ proves $M >_{MUL} N$. □

Theorem 3.9.
If $>$ is a strict order, then $>_{MUL}$ and $>_{mul}$ are also strict orders.

Proof. By Theorem 3.7 for orders $>_{MUL} = >_{mul}$. We continue by proving that $>_{MUL}$ is a strict order. We get transitivity by Lemma 3.6. For irreflexivity, by Lemma 3.8, we get $M >_{MUL} M \implies \emptyset >_{MUL} \emptyset$ but by definition $\emptyset$ is a minimal element of $>_{MUL}$. □

3.3 Well-foundedness of Multiset Extensions of a Relation

In this section we will prove that two variants of multiset extension introduced before preserve well-foundedness, so a multiset extension of a well-founded relation is again well-founded. The proof, by well-founded part induction (see Section 2), follows [28]. We will abbreviate $\mathcal{W}^{\mathbb{M}_A}_{\lhd_{mul}}$ by $\mathcal{W}^{\mathbb{M}}_{\lhd}$. By $\lhd_{mul}$ we denote transposition of $\rhd_{mul}$; similarly for $<_{mul}$ and $<_{MUL}$. We begin with a simple auxiliary lemma considering $\lhd_{mul}$.

Lemma 3.10.
Let $M, N \in \mathbb{M}_A$. If $N \lhd_{mul} M \cup \{\{a\}\}$ then there exist a multiset $M' \in \mathbb{M}_A$ such that:

$$\left(\begin{array}{c} N = M' \cup \{\{a\}\} \\ M' \lhd_{mul} M \end{array}\right) \vee \left(\begin{array}{c} N = M \cup M' \\ \forall x \in M' \ . \ x < a \end{array}\right)$$

Proof. Easy from the definition of $\lhd_{mul}$ (see [23] for a detailed proof). □

Lemma 3.11.
Let $M_0 \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ and $a \in A$. Then:

$$\left.\begin{array}{ll} (1) & \forall b < a, M \in \mathcal{W}^{\mathbb{M}}_{\lhd} \ . \ M \cup \{\{b\}\} \in \mathcal{W}^{\mathbb{M}}_{\lhd} \\ (2) & \forall M \lhd_{mul} M_0 \ . \ M \cup \{\{a\}\} \in \mathcal{W}^{\mathbb{M}}_{\lhd} \end{array}\right\} \implies M_0 \cup \{\{a\}\} \in \mathcal{W}^{\mathbb{M}}_{\lhd}$$

Proof. By the definition of $\mathcal{W}^{\mathbb{M}}_{\lhd}$ we need to show $N \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ for $N \lhd_{mul} M_0 \cup \{\{a\}\}$. By Lemma 3.10 there are two possibilities for $N \lhd_{mul} M_0 \cup \{\{a\}\}$:

- $N = M \cup \{\{a\}\}$ for some $M \lhd_{mul} M_0$. Then $N \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ by (2).
- $N = M_0 \cup K$ and $\forall k \in K \ . \ k < a$. We proceed by induction on K. For base case if $K = \emptyset$ then $N = M_0 \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ by assumption. For induction step $K = K_0 \cup \{\{k\}\}$ and $K_0 \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ by the induction hypothesis. So $N \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ follows from (1) as $k < a$.

□

Lemma 3.12.

$$\forall b < a, M \in \mathcal{W}^{\mathbb{M}}_{\lhd} \;.\; M \cup \{\{b\}\} \in \mathcal{W}^{\mathbb{M}}_{\lhd} \implies \forall M \in \mathcal{W}^{\mathbb{M}}_{\lhd} \;.\; M \cup \{\{a\}\} \in \mathcal{W}^{\mathbb{M}}_{\lhd}$$

Proof. Follows immediately from Lemma 3.11 by well-founded part induction on M. □

Lemma 3.13.
$\forall a \in \mathcal{W}^{A}_{<}, M \in \mathcal{W}^{\mathbb{M}}_{\lhd} \;.\; M \cup \{\{a\}\} \in \mathcal{W}^{\mathbb{M}}_{\lhd}$

Proof. Follows immediately from Lemma 3.12 by well-founded induction on a. □

Lemma 3.14.
$\forall M \in \mathbb{M}_A \;.\; M \in \mathcal{W}^{\mathbb{M}}_{\lhd}$

Proof. By induction on M. For base case $\emptyset \in \mathcal{W}^{\mathbb{M}}_{\lhd}$ as there is no multiset N such that $N \lhd_{mul} \emptyset$. The induction step follows from Lemma 3.13. □

Theorem 3.15.
If $<$ is a well-founded relation then its multiset extensions $<_{mul}$ and $<_{MUL}$ are also well-founded.

Proof. $<_{mul}$ is well-founded as it is a transitive closure of $\lhd_{mul}$ which is well-founded by Lemma 3.14. Then well-foundedness of $<_{MUL}$ follows from Lemma 3.2. □

Coq

Nipkow remarked that the variant of a proof of well-foundedness of the multiset extension from [28] is particularly well suited for theorem provers. Indeed this proof went quite smoothly in Coq and it is rather short.

Chapter 4

Simply Typed λ-calculus

The lambda calculus (λ-calculus) was introduced by Church and Kleene in 1930s. Initially Church intended to use it as a formal system for the foundations of mathematics. The full system turned out to be inconsistent but the λ-calculus became a successful and widely used model of computations.

In this chapter we will present a formalization of the simply-typed λ-calculus ($\lambda^{\rightarrow}$) – a variant of typed λ-calculus with $\rightarrow$ as the only type constructor. We will present its version with constants and with typing à la Church.

We will define terms in Section 4.1, then we will introduce some further definitions and results in Section 4.2. Section 4.3 will be devoted to the definition of a many-variable, typed substitution and Section 4.4 to development of an equivalence relation on terms that extends the concept of α-convertibility to free variables. Finally Section 4.5 introduces the β-reduction relation along with its properties. For a more detailed introduction to simply typed lambda calculus we refer to, for instance, [12, 6].

4.1 Definition of Terms

We assume a set of sorts (ground types) and we define simple types.

Definition 4.1 (Simple types, $\mathcal{T}_\mathcal{S}$).
Assume a set of sorts $\mathcal{S}$*. We inductively define a set of* simple types $\mathcal{T}_\mathcal{S}$ *as follows:*

- *for any* $\alpha \in \mathcal{S}$, $\alpha \in \mathcal{T}_\mathcal{S}$ (base type)*,*
- *if* $\alpha, \beta \in \mathcal{T}_\mathcal{S}$ *then* $\alpha \rightarrow \beta \in \mathcal{T}_\mathcal{S}$ (arrow type)*.*

We will denote simple types by α, β, δ *etc.*

Now we can define a notion of signature.

Definition 4.2 (Signature, Σ).
We assume a set of constants $\mathcal{F}$ *and we define a* signature *(*Σ*) to be a set of typed constants declarations that is pairs* $f : \alpha$ *with* $f \in \mathcal{F}$ *and* $\alpha \in \mathcal{T}_\mathcal{S}$*.*

We will usually refer to typed constants as function symbols *and we will denote them by* f, g *etc.*

For the rest of the presentation we assume a set of sorts $\mathcal{S}$ and a signature Σ to be fixed and we assume a fixed set of variables $\mathcal{V}$.

We define environments to hold declarations for free variables.

Definition 4.3 (Environment, $\mathcal{E}nv$).
Environment *($\mathcal{E}nv$) is defined as a finite set of distinct variable declarations, that is: $\mathcal{E}nv \subset \mathcal{V} \times \mathcal{T}_{\mathcal{S}}$ such that for every environment $\Gamma : \mathcal{E}nv$, $\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$ for all $1 \leq i, j \leq n$: $x_i \neq x_j$ for $i \neq j$. The domain of the environment is defined as $Var(\Gamma) = \{x_1, \ldots, x_n\}$.*

We will denote environments as Γ, Δ etc.

Now we define un-typed terms which we will call preterms.

Definition 4.4 (Preterms, $\mathcal{P}t$).
A set of preterms *is defined by the following grammar:*

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

The grammar rules for preterms define respectively: a variable, a function symbol, an application and an abstraction.

Now we proceed with presenting typing judgements: a typing discipline that our typed terms will follow.

Definition 4.5 (Typing judgements).
We will write typing judgements *of the form $\Gamma \vdash t : \alpha$ to denote that in an environment Γ a preterm t has type α. They respect the rules of the following inference system:*

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

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

Definition 4.6 (Typed terms, Λ).
Typed terms *(Λ) are identified with typing judgements:*

$$\Lambda = \{\Gamma \vdash t : \alpha \mid \Gamma \in \mathcal{E}nv, t \in \mathcal{P}t, \alpha \in \mathcal{T}_{\mathcal{S}}\}$$

We also define:

$$\begin{aligned} \mathsf{env}(\Gamma \vdash t : \alpha) &= \Gamma \\ \mathsf{term}(\Gamma \vdash t : \alpha) &= t \\ \mathsf{type}(\Gamma \vdash t : \alpha) &= \alpha \end{aligned}$$

We will denote typed terms by letters t, u etc. Often we will omit the environments and write $t : \alpha$ instead of $\Gamma \vdash t : \alpha$ or even only t if the type is irrelevant.

Coq

The development of $\lambda^{\to}$ in Coq is modularized using Coq's module mechanism. Firstly in file TermsSig.v the module type SimpleTypes is defined containing definition of simple types parameterized by the set of ground types.[1]

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

[1]The presentation of Coq scripts in this paper is slightly simplified for the sake of readability. Interested reader is encouraged to compare with the actual Coq scripts.

Then the module Signature builds on that and contains definition of constants along with the 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 development concerning $\lambda^{\rightarrow}$ is done within functors taking such signature as their argument. In the file TermsDef.v the definition of typed terms is given.

Note that we use de Bruijn indices [11] to represent terms in order to avoid having to explicitly deal with α-conversion. Due to that fact the environments could simply be represented by lists of simple types. However, later on we will introduce a lifting operation on terms that renames variables, which in case of de Bruijn indices corresponds to increasing the numerical values of variables. This leaves some variables with lower indexes undeclared and to express that fact we introduce dummy variables. So environments are lists of SimpleType option, and not SimpleType, with None representing dummy variables. However this will lead to some small complications as we will explain in Section 4.2.1. We will also use the notation `E |= x := A` to denote that the variable `x` in the environment `E` has type `A` and `E |= x :!` to denote that the variable `x` is undeclared in the environment `E`.

```
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).
```

We continue with straightforward definition of preterms. Note that variables are natural numbers representing their index in de Bruijn notation. 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 "s [ x ]" := (s @@ x) (at level 30).
Notation "s [ x, y ]" := (s @@ x @@ y) (at level 30).
Notation "\ A => M" := (Abs A M) (at level 35).
```

We present typing judgements next. Typing judgements will be written in Coq in the form `E |- M := A` representing typing judgement $E \vdash M : A$. It is easy to recognize the inference system from the Definition 4.5 in the following inductive 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 the definition of typed term. Typed term consist of an environment, a preterm, a type and a typing judgement certifying that in the given environment, the given preterm has given type.

```
Record Term : Set := buildT {
  env:    Env;
  term:   Preterm;
  type:   SimleType;
  typing: Typing env term type
}.
```

Definitions introduced in this section are very crucial as we will constantly work with them while formalizing the content of the following sections. One may wonder whether it would not be more convenient to use a single dependent inductive definition of typed terms that would combine the definition of term structure with its typing judgement and that could look as follows:

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

At first sight this definition looks very attractive but although having some advantages, it also has a serious drawback; namely the fact that the structure of terms is embedded within its typing judgement. As we shall see later the great part of the proofs of equality of two terms will use the observation, that is to be proven in Section 4.2.2, that two terms with equal structure and equal environments are equal. Such proofs essentially split the reasoning into the reasoning about term structure (un-typed λ-terms) and about environments. This is very convenient to do with the use of the first proposed definition as it requires only dealing with two very simple definitions (Env and Preterm) as opposed to the second approach where one needs to constantly work with a complex dependent type. Moreover in this way all the reasoning about term structures gives us some results about the theory of un-typed λ-calculus.

4.2 Further Properties and Definitions of $\lambda^{\rightarrow}$

4.2.1 Environment Properties

We start with some simple operations on environments:

Definition 4.7 (Environment operations).
For any environments Γ, Δ we define the binary operations of composition and subtraction of environments.

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

We also introduce a notion of compatibility of environments.

Definition 4.8 (Environment compatibility).
For environments Γ, Δ we say that they are compatible iff for any variable v if they both declare it, they declare it with the same type. We will denote the fact that Γ is compatible with Δ by $\Gamma \leftrightsquigarrow \Delta$. So we have:

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

Coq

All the results concerning environments can be located in the file TermsEnv.v. Below we present definitions of environment composition and subtraction from Definition 4.7.

```
Fixpoint env_compose (E F: Env) {struct E} : Env :=
  match E, F with
  | nil, nil => EmptyEnv
  | L, nil => L
  | nil, L => L
  | _::E', Some a::F' => Some a :: env_compose E' F'
  | e::E', None::F' => e :: env_compose E' F'
  end.
Notation "E [+] F" := (env_compose E F) (at level 50, left associativity).

Fixpoint env_subtract (E F: Env) {struct E} : Env :=
  match E, F with
  | nil, _ => EmptyEnv
  | E, nil => E
  | None :: E', _ :: F' => None :: env_subtract E' F'
  | Some e :: E', None :: F' => Some e :: env_subtract E' F'
  | Some e :: E', Some f :: F' => None :: env_subtract E' F'
  end.
Notation "E [-] F" := (env_subtract E F) (at level 50, left associativity).
```

The definition of environment compatibility (Definition 4.8) in Coq looks as follows:

```
Definition env_comp_on E F x : Prop :=
  forall A B,
    E |= x := A ->
    F |= x := B ->
    A = B.

Definition env_comp E F : Prop :=
  forall x, env_comp_on E F x.
Notation "E [<->] F" := (env_comp E F) (at level 70).
```

We already indicated in Section 4.1, while introducing terms, that allowing dummy variables in environments (which will be useful in Section 4.3, in particular in 4.3.2) leads to problems. Those problems come from the fact that in this way we loose unique representation of environment. For instance empty environment can be represented by the empty list (`nil`) but also by a list with only a single declaration for dummy (`None::nil`). This problem was solved by providing equality for environments, different than `Coq`'s Leibniz' equality, that takes those subtle representation issues into account. It is define via subset predicate for environments, that is $\Gamma = \Delta \iff \Gamma \subseteq \Delta \wedge \Delta \subseteq \Gamma$.

```
Definition envSubset E F :=
  forall x A, E |= x := A -> F |= x := A.

Definition env_eq E1 E2 := envSubset E1 E2 /\ envSubset E2 E1.
Notation "E1 [=] E2" := (env_eq E1 E2) (at level 70).
```

4.2.2 Typing Properties

Now we will look into some properties of the type system of the simply typed λ-calculus. The first two properties ensures that every term has a unique type and that type derivations are unique.

Theorem 4.1 (Uniqueness of types).
Suppose $\Gamma \vdash t : \alpha$ *and* $\Gamma \vdash t : \beta$ *then* $\alpha = \beta$.

Proof. Easy structural induction on t. □

Theorem 4.2 (Uniqueness of typing judgements).
Given term $\Gamma \vdash t : \alpha$ *its type derivation is unique.*

Proof. Easy structural induction on t. □

Now we present a theorem stating that typability of simply typed λ-terms is decidable in linear time.

Theorem 4.3 (Decidability of typing).
Given environment Γ *and preterm* t *the problem of finding* α *such that* $\Gamma \vdash t : \alpha$ *is decidable in linear time with respect to the sum of the sizes of* t *and* Γ.

Proof. Induction on the structure of t. □

Two following lemmas express the fact that a term can be typed in an extended environment. We will need them in Section 4.3.3 for reasoning about substitution.

Lemma 4.4.
If $\Gamma \vdash t : \alpha$ *then for any environment* Δ, $\Delta \cdot \Gamma \vdash t : \alpha$.

Proof. Easy structural induction on t. □

Lemma 4.5.
If $\Gamma \vdash t : \alpha$ *then for any environment* $\Delta \leftrightsquigarrow \Gamma$, $\Gamma \cdot \Delta \vdash t : \alpha$.

Proof. Easy structural induction on t. For the variable case we use compatibility of Δ with Γ. □

Coq

Few remarks:

- The proof of Theorem 4.2 is actually not so easy in Coq. It involves the usage of the uniqueness of identity proofs for dependent types and makes use of Streicher's axiom K from the standard library. I would like to express my gratitude to Roland Zumkeller who helped me carry out this proof.
- From the constructive Coq proof of Theorem 4.3 a certified algorithm for typing lambda terms can be extracted easily.

4.2.3 Further Definitions

We define a set of free variables of a term as:

Definition 4.9 (Free variables, Vars).
For $\Gamma \vdash t : \alpha$ we define the environment containing free variables of t, $\mathsf{Vars}(t)$, by induction on t as:

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

We define a subterm relation as follows:

Definition 4.10 (Sub-term, $\sqsubset$, $\sqsubseteq$).
The subterm *($\sqsubseteq$) and* strict subterm *($\sqsubset$) relations are inductively defined as:*

$$\frac{t \sqsubseteq u_l}{t \sqsubset @(u_l, u_r)} \qquad \frac{t = u}{t \sqsubseteq u} \qquad \frac{t \sqsubseteq u_r}{t \sqsubset @(u_l, u_r)} \qquad \frac{t \sqsubset u}{t \sqsubseteq u} \qquad \frac{t \sqsubseteq u}{t \sqsubset \lambda x\!:\!\alpha.u}$$

We proceed with showing that strict subterm relation is well-founded. This result justifies the use of induction on the structure of terms by which we mean induction with respect to the $\sqsubset$ relation on terms and which will be frequently used in the subsequent proofs.

Theorem 4.6 (Well-foundedness of $\sqsubset$).
$\sqsubset$ is a well-founded relation.

Proof. We need to prove $t \in \mathcal{A}cc$ for every term t which can easily be shown by structural induction on t. □

The following two notions will be used in the definition of HORPO in Chapter 7.

Definition 4.11 (Partial left-flattening).
Given term $@(t_1, \ldots, t_n)$, a list of terms $@(t_1, t_2, \ldots, t_i), t_{i+1}, \ldots, t_n$ is called its partial left-flattening, *for $1 \leq i \leq n$.*

Definition 4.12 (Neutral term).
A term t is called neutral *if it is not an abstraction.*

Coq

In Coq development an environment containing only free variables and no additional unused declarations is called an active environment of a term and is defined as follows by a recursion on term structure. The recursion is hidden in the use of `Typing_rec` which is an induction principle generated by Coq for type `Typing` representing typing judgements.

```
Definition activeEnv (M: Term) : Env :=
  match M with
  | buildT E Pt T M0 =>
    Typing_rec
    (fun E0 Pt0 T0 _ => Env)
    (fun _ x A _ => copy x None ++ A [#] EmptyEnv)
    (fun _ _ => EmptyEnv)
    (fun _ _ _ _ _ _ Ein => tail Ein)
    (fun _ _ _ _ _ _ _ El _ Er => El [+] Er)
    M0
  end.
```

Then a number of auxiliary functions is provided. Functions `appBodyL`, `appBodyR` and `absBody` return the left argument of an application, the right argument of an application and the body of an abstraction respectively and are defined as expected. Predicates `isVar`, `isFunS`, `isAbs` and `isApp` hold for a term that is a variable, a function symbol, an abstraction or an application respectively. For an application $t = @(t_1, t_2, \dots, t_n)$ we call t_1 an application head, $t_2, \dots, t_n$ application arguments and $t_1, \dots, t_n$ application units. `appHead t` returns the head of t; `isArg t' t` and `isAppUnit t' t` hold if t' is an application argument or application unit, respectively.

The subterm relations corresponding to mutually recursive definitions of $\sqsubset$ and $\sqsubseteq$ are as follows:

```
Inductive subterm: Term -> Term -> Prop :=
| AppLsub: forall M N (Mapp: isApp M),
    subterm_le N (appBodyL Mapp) ->
    subterm N M
| AppRsub: forall M N (Mapp: isApp M),
    subterm_le N (appBodyR Mapp) ->
    subterm N M
| Abs_sub: forall M N (Mabs: isAbs M),
    subterm_le N (absBody Mabs) ->
    subterm N M
 with subterm_le: Term -> Term -> Prop :=
| subterm_lt: forall M N,
    subterm N M ->
    subterm_le N M
| subterm_eq: forall M,
    subterm_le M M.
```

4.3 Substitution

In this section we introduce the substitution on terms of $\lambda^{\rightarrow}$. First in 4.3.1 we introduce concepts not directly related to substitution, namely those of positions in a term and the replacement operation. In 4.3.2 we define lifting and lowering operations on terms, which will be used in the definition of substitution. Finally in 4.3.3 we define substitution on $\lambda^{\rightarrow}$ terms and discuss some of its properties.

4.3.1 Positions and Replacement

We begin by defining term positions.

Definition 4.13 (Term positions, $\mathcal{P}os_{\Lambda}$).
We define positions *($\mathcal{P}os$) as strings over the following set:*

$$\{\epsilon, \lhd, \rhd, \lambda\}$$

its elements indicating position at the root, position in left and right argument of an application and position within a lambda abstraction respectively.

Now we inductively define term positions *($\mathcal{P}os_\Lambda$) as a family of positions indexed by terms, as follows:*

$$\frac{t \in \Lambda}{\epsilon \in \mathcal{P}os_t} \qquad \frac{p \in \mathcal{P}os_t}{\lambda \cdot p \in \mathcal{P}os_{\lambda x:\alpha.t}}$$

$$\frac{p \in \mathcal{P}os_{t_l}}{\lhd \cdot p \in \mathcal{P}os_{@(t_l,t_r)}} \qquad \frac{p \in \mathcal{P}os_{t_r}}{\rhd \cdot p \in \mathcal{P}os_{@(t_l,t_r)}}$$

We continue with definition of subterm at position and replacement of term at position.

Definition 4.14 (Subterm at position, $t|_p$)**.**
For any term $\Gamma \vdash t : \alpha$ and position $p \in \mathcal{P}os_t$ we give a recursive definition of a subterm of t at p, *$t|_p$:*

$$\begin{array}{rcl} t|_\epsilon & = & t \\ @(t_l,t_r)|_{\lhd \cdot p} & = & t_l|_p \\ @(t_l,t_r)|_{\rhd \cdot p} & = & t_r|_p \\ \lambda x\!:\!\alpha.t|_{\lambda \cdot p} & = & t|_p \end{array}$$

Definition 4.15 (Replacement at position, $t[u]_p$)**.**
For any term $\Gamma \vdash t : \alpha$, position $p \in \mathcal{P}os_t$ and term $\Delta \vdash u : \beta$ such that $\mathsf{type}(t|_p) = \beta$ and $\mathsf{env}(t|_p) = \Delta$ we define replacement in term t at position p with term u *($t[u]_p$), by recursion on p, as:*

$$\begin{array}{rcl} t[u]_\epsilon & = & u \\ @(t_l,t_r)[u]_{\lhd \cdot p} & = & @(t_l[u]_p, t_r) \\ @(t_l,t_r)[u]_{\rhd \cdot p} & = & @(t_l, t_r[u]_p) \\ \lambda x\!:\!\alpha.t[u]_{\lambda \cdot p} & = & \lambda x\!:\!\alpha.t[u]_p \end{array}$$

Proposition 4.7.
For any terms $\Gamma \vdash t : \alpha$, $\Delta \vdash u : \beta$ and position $p \in \mathcal{P}os_t$ such that the replacement $t[u]_p$ is well defined we have:

$$\Gamma \vdash t[u]_p : \alpha$$

so the result of a replacement is typable with the same environment and type as the term in which the replacement takes place.

Proof. Induction no p.

- $p = \epsilon$. Then $t[u]_p = u$, $\Gamma = \mathsf{env}(t|_\epsilon) = \mathsf{env}(t) = \Delta$ and $\beta = \mathsf{type}(t|_\epsilon) = \mathsf{type}(t) = \alpha$ by definition and then $\Gamma \vdash t[u]_p : \alpha$ as $\Delta \vdash u : \beta$.
- $p = \lambda \cdot p'$, then $t = \lambda x : \alpha.t'$ and $t[u]_p = \lambda x : \alpha.t'[u]_{p'}$ and we conclude by the induction hypothesis for $t'[u]_{p'}$.
- $p = \lhd \cdot p'$, then $t = @(t_l, t_r)$ and $t[u]_p = @(t_l[u]_{p'}, t_r)$ and we conclude by the induction hypothesis for $t_l[u]_{p'}$.
- $p = \rhd \cdot p'$, then $t = @(t_l, t_r)$ and $t[u]_p = @(t_l, t_r[u]_{p'})$ and we conclude by induction hypothesis for $t_r[u]_{p'}$.

□

Coq

The headers of definitions of term positions (Definition 4.13), subterm at position (Definition 4.14) and replacement (Definition 4.15) are presented below. The definitions are rather standard and hence not included here.

```
Inductive Pos : Term -> Set := (...)
Fixpoint termAtPos (M: Term) (pos: Pos M) : Term := (...)
Notation "M // pos" := (@termAtPos M pos) (at level 40).
Fixpoint swap_term (M: Term) (pos: Pos M) (R: Term) : Preterm := (...)
Definition PlaceHolder M pos :=
  { T: Term |
      env (M // pos) = env T &
      type (M // pos) = type T
  }.
Definition swap_aux (M: Term) (pos: Pos M) (R: PlaceHolder pos) :
  {N: Term |
     env N = env M /\
     type N = type M /\
     term N = swap_term pos (proj1_sig2 R)
  }.
Proof
   (...)
Defined.
Definition swap M pos N : Term :=
  proj1_sig (swap_aux (M := M) (pos := pos) N).
```

4.3.2 Lifting and Lowering of Terms

So far we tried to hide the use of de Bruijn indices from presentation but it is not possible it this section as lifting and lowering are operations specifically defined for terms using this representation. So throughout this section we assume $\mathcal{V} = \mathbb{N}$.

Before we define those operations let us briefly explain why do we need them. In the process of substitution we replace a term in some context which may contain binders. To avoid capturing of free variables their de Bruijn indices need to be increased by a value equal to the number of binders in the context in which a substitution takes place. This operation is called a lifting of a term. Lowering is the opposite operation in which the indices are decreased.

Definition 4.16 (Term lifting, $t{\uparrow}^n_k$).
For a term t and $n, k \in \mathbb{N}$ we define its lifted version with variables with index less than k untouched and remaining ones increased by n ($t{\uparrow}^n_k$) as:

$$\begin{array}{rcll} f{\uparrow}^n_k & = & f & \\ x{\uparrow}^n_k & = & x & \textit{if } x < k \\ x{\uparrow}^n_k & = & x + n & \textit{if } x \geq k \\ @(t_l, t_r){\uparrow}^n_k & = & @(t_l{\uparrow}^n_k, t_r{\uparrow}^n_k) & \\ \lambda x\!:\!\alpha.t{\uparrow}^n_k & = & \lambda x\!:\!\alpha.t{\uparrow}^n_{k+1} & \end{array}$$

We also define $t{\uparrow}^n := t{\uparrow}^n_0$.

Similarly we define lifting of environments:

Definition 4.17 (Environment lifting, $\Gamma{\uparrow}^n_k$).
For an environment $\Gamma = \{x_1\!:\!\alpha_1, \ldots, x_n\!:\!\alpha_n\}$ we define its lifted version $\Gamma{\uparrow}^n_k$ as:

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

We also define $\Gamma{\uparrow}^n := \Gamma{\uparrow}^n_0$.

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

Proposition 4.8.
If $\Gamma \vdash t : \alpha$ then for any $n, k \in \mathbb{N}$: $\Gamma{\uparrow}^n_k \vdash t{\uparrow}^n_k : \alpha$.

Proof. Structural induction on t:

- $t = x$. Either $x < k$ or $x \geq k$ but in both cases $\Gamma\uparrow^x_k \vdash x\uparrow^n_k : \alpha$ by definitions of preterm and environment lifting.
- $t = f$. Then $f\uparrow^n_k = f$ and $\Gamma\uparrow^n_k \vdash f\uparrow^n_k : \alpha$ by typing rule for constant.
- $t = \lambda x : \beta.t_b$ with $t_b : \xi$ and $\alpha = \beta \to \xi$. We conclude $\Gamma\uparrow^n_k \vdash (\lambda x : \beta.t_b)\uparrow^n_k : \alpha$ by induction hypothesis for $t_b\uparrow^n_{k+1}$ and by typing rule for abstraction.
- $t = @(t_l, t_r)$. We conclude $\Gamma\uparrow^n_k \vdash @(t_l, t_r)\uparrow^n_k : \alpha$ by induction hypothesis for $t_l\uparrow^n_k$, induction hypothesis for $t_r\uparrow^n_k$ and by typing rule for application.

$\square$

The definitions and results for lowering are dual except that this time we need to take care of not lowering indices below 0.

Definition 4.18 (Term lowering, $t\downarrow^n_k$).
For a term t and $n, k \in \mathbb{N}$ we define its lowered version with variables with index less than k untouched and remaining ones decreased by n ($t\downarrow^n_k$) as:

$$\begin{array}{rcll} f\downarrow^n_k & = & f & \\ x\downarrow^n_k & = & x & \textit{if } x < k \\ x\downarrow^n_k & = & x \ominus n & \textit{if } x \geq k \\ @(t_l, t_r)\downarrow^n_k & = & @(t_l\downarrow^n_k, t_r\downarrow^n_k) & \\ \lambda x{:}\alpha.t\downarrow^n_k & = & \lambda x{:}\alpha.t\downarrow^n_{k+1} & \end{array}$$

where $m \ominus n := max(0, m - n)$.

We also define $t\downarrow^n := t\downarrow^n_0$.

Definition 4.19 (Environment lowering, $\Gamma\downarrow^n_k$).
For an environment $\Gamma = \{x_1{:}\alpha_1, \ldots, x_n{:}\alpha_n\}$ we define its lowered version $\Gamma\downarrow^n_k$ as:

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

We also define $\Gamma\downarrow^n := \Gamma\downarrow^n_0$.

We have similar result to that of Proposition 4.8 just this time we need to make sure that in the initial term indices $k, \ldots, k + n - 1$ are unused.

Proposition 4.9.
For any $n, k \in \mathbb{N}$ and any term $\Gamma \vdash t : \alpha$ if $\forall i \in \{k, \ldots, k+n-1\}$. $x_i \notin Var(\Gamma)$ then $\Gamma\downarrow^n_k \vdash t\downarrow^n_k : \alpha$.

Proof. Structural induction on t. The proof is similar to the proof of Proposition 4.8. We use the assumption that $x_i \notin Var(\Gamma)$ for $k \leq i < k + n$ in the variable case to ensure that by lowering no variable declarations are lost. $\square$

Coq

Lifting on preterms in Coq is a straightforward translation of Definition 4.16 and is defined as:

```
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.
```

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

- `initialSeg l n` returns first 'n' elements of a list 'l' (or less if the list is shorter).
- `finalSeg l n` returns the suffix of 'l' starting at a 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 by 'n' the indices of all the variables bigger than 'k'.

Then the lifting on typed terms, corresponding to Proposition 4.8, is defined as:

```
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).
```

The somewhat dual definitions of lowering follow. Note that only variant of lowering by one is provided as a general version of lowering was not needed for the development (but can be obtained by application of lowering by one a number of times).

```
Fixpoint prelower_aux (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 (pred i)
      | right _ => (* i < k *)  Var i
      end
  | App M N => App (prelower_aux M k) (prelower_aux N k)
  | Abs A M => Abs A (prelower_aux M (S k))
  end.
Definition prelower P := prelower_aux P 0.

Definition loweredEnv (E: Env) (k: nat) : Env :=
  initialSeg E k ++ finalSeg E (S k).
```

```
Definition lower_aux (M: Term) (k: nat):
  env M |= k :! ->
  {N: Term |
     env N = loweredEnv (env M) k /\
     term N = prelower_aux (term M) k /\
     type N = type M
  }.
Proof.
  (...)
Defined.

Definition lower (M: Term) (ME: env M |= 0 :!) : Term :=
  proj1_sig (lower_aux M ME).
```

4.3.3 Definition of Substitution

Now we can present the definition of substitution. First we present it for un-typed terms.

Definition 4.20 (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 the substitution γ with its domain restricted to $\mathcal{V} \setminus \mathcal{X}$, that is:

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

Definition 4.21 (Substitution on preterms)**.**
We define substitution on preterms *as follows:*

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

Note that computation $\gamma_{\setminus\{x\}}$ in de Bruijn notation is realized via taking lifted version of γ: $\gamma\uparrow^1$, with lifting operation on substitution defined as follows:

Definition 4.22 (Substitution lifting)**.**
Let $\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \ldots, x_n/\Gamma_n \vdash t_n : \alpha_n\}$ be a substitution. We define its lifted version as:

$$\gamma\uparrow^n = \{(x_1 + n)/(\Gamma_1 \vdash t_1 : \alpha_1)\uparrow^n, \ldots, (x_n + n)/(\Gamma_n \vdash t_n : \alpha_n)\uparrow^n\}$$

Substitution operates on typed terms and hence is not always applicable as there may be type and environment clashes. The following definition captures condition that are required for a substitution to be applicable to a term.

Definition 4.23 (Compatibility of substitution)**.**
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:*
  $\forall i \neq j \in \{1,\dots,n\}$. $\Gamma_i \leftrightsquigarrow \Gamma_j$.
- *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 following result ensures that the conditions posted in the above definition are sufficient to type the result of application of substitution. This is a stronger version of the result from [20]. We need two auxiliary lemmas first.

Lemma 4.10.
Let $\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \dots, x_n/\Gamma_n \vdash t_n : \alpha_n\}$ be a substitution such that all terms in γ have compatible environments, that is: $\forall i \neq j \in \{1,\dots,n\}$. $\Gamma_i \leftrightsquigarrow \Gamma_j$. Then for any i, $\mathit{Ran}(\gamma) = \mathit{Ran}(\gamma) \cdot \Gamma_i$.

Proof. The result follows from the fact that environment composition is idempotent and commutative for compatible environments. □

Lemma 4.11.
Let $\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \dots, x_n/\Gamma_n \vdash t_n : \alpha_n\}$ be a substitution such that all terms in γ have compatible environments, that is: $\forall i \neq j \in \{1,\dots,n\}$. $\Gamma_i \leftrightsquigarrow \Gamma_j$. Then for any i, $\mathit{Ran}(\gamma) \vdash t_i : \alpha_i$.

Proof. By Lemma 4.10 $\mathit{Ran}(\gamma) = \mathit{Ran}(\gamma) \cdot \Gamma_i$ and then $\mathit{Ran}(\gamma) \cdot \Gamma_i \vdash t_i : \alpha_i$ by Lemma 4.4. □

Theorem 4.12.
Let $\Gamma \vdash t : \alpha$ be a term and let $\gamma = \{x_1/\Gamma_1 \vdash t_1 : \alpha_1, \dots, x_n/\Gamma_n \vdash t_n : \alpha_n\}$ be a substitution compatible with this term. Then:

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

Proof. Structural induction on t.

- $t = x$.
  - If $x \in \mathit{Dom}(\gamma)$ then $x/\Gamma_i \vdash t_i : \alpha_i \in \gamma$ for some i and $x\gamma = t_i$. By Lemma 4.11 we get that $\mathit{Ran}(\gamma) \vdash t_i : \alpha_i$, by Lemma 4.4 $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash t_i : \alpha_i$ and finally $\alpha_i = \alpha$ by the assumption on compatibility of γ with t (hence domain of γ is compatible with t).
  - If $x \notin \mathit{Dom}(\gamma)$ then $x\gamma = x$ and to conclude the result by the typing rule for variable we need to show that $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash x : \alpha$. Either $x : \beta \in \mathit{Ran}(\Gamma)$ but then $\beta = \alpha$ by compatibility of Γ with $\mathit{Ran}(\gamma) \setminus \mathit{Dom}(\gamma)$ (γ is compatible with t; note that $x \notin \mathit{Dom}(\gamma)$). Or $x \notin \mathit{Var}(\mathit{Ran}(\Gamma))$ but then $x : \alpha \in \Gamma$ and hence $x : \alpha \in (\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma)$.
- $t = f$. Then $f\gamma = f$ and $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash f : \alpha$ by the typing rule for constant.
- $t = @(t_l, t_r)$ with $\Gamma \vdash t_l : \xi \to \alpha$ and $\Gamma \vdash t_r : \xi$. Then $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash t_l\gamma : \xi \to \alpha$ and $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash t_r\gamma : \xi$ by the induction hypothesis and $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash @(t_l, t_r)\gamma : \alpha$ by the typing rule for application as $@(t_l, t_r)\gamma = @(t_l\gamma, t_r\gamma)$.
- $t = \lambda x : \beta . t_b$. Substitution γ is compatible with $\Gamma \vdash t : \alpha$ so $\Gamma \leftrightsquigarrow \mathit{Dom}(\gamma)$ and hence $\Gamma \cdot \{x : \beta\} \leftrightsquigarrow \mathit{Dom}(\gamma_{\setminus\{x\}})$. Similarly $\Gamma \leftrightsquigarrow \mathit{Ran}(\gamma) \setminus \mathit{Dom}(\gamma)$ and hence $\Gamma \cdot \{x : \beta\} \leftrightsquigarrow \mathit{Ran}(\gamma_{\setminus\{x\}}) \setminus \mathit{Dom}(\gamma_{\setminus\{x\}})$. So substitution $\gamma_{\setminus\{x\}}$ is compatible with the term $\Gamma \cdot \{x : \beta\} \vdash t_b : \beta \to \alpha$ and by the induction hypothesis we get $(\Gamma \cdot \{x : \beta\} \setminus \mathit{Dom}(\gamma_{\setminus\{x\}})) \cdot \mathit{Ran}(\gamma_{\setminus\{x\}}) \vdash t_b\gamma_{\setminus\{x\}} : \beta \to \alpha$. By the typing rule for abstraction and by observation that $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \cdot \{x : \beta\} = (\Gamma \cdot \{x : \beta\} \setminus \mathit{Dom}(\gamma_{\setminus\{x\}})) \cdot \mathit{Ran}(\gamma_{\setminus\{x\}})$ we conclude $(\Gamma \setminus \mathit{Dom}(\gamma)) \cdot \mathit{Ran}(\gamma) \vdash \lambda x : \beta . t_b\gamma : \alpha$.

□

Coq

Part concerning substitution is by far the largest part of the development of $\lambda^{\rightarrow}$. That is primarily because indeed the definition of substitution on typed terms is rather complex. But also the development contains some more results about substitution not included in this presentation.

Because of the use of de Bruijn indices substitution is simply a list of `option Term` (`None` indicating that given index is not in the domain of given substitution). The definition along with some notations:

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

Definition varSubstTo (G: Subst) x T : Prop :=
  nth_error G x = Some (Some T).
Notation "G |-> x / T" := (varSubstTo G x T) (at level 50, x at level 0).
```

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

```
Definition subst_dom (G: Subst) : Env :=
  map (fun T =>
    match T with
    | None => None
    | Some T => Some (type T)
    end) G.

Definition subst_ran (G: Subst) : Env :=
  fold_left (fun E S =>
    match S with
    | None => E
    | Some T => E [+] env T
    end
  ) G EmptyEnv.
```

The definition of compatibility of asubstitution with a term:

```
Definition subst_envs_comp (G: Subst) : Prop :=
  forall i j Ti Tj,
    G |-> i/Ti ->
    G |-> j/Tj ->
    env Ti [<->] env Tj.

Record correct_subst (M: Term) (G: Subst) : Prop := {
  envs_c:
    subst_envs_comp G;
  dom_c:
    subst_dom G [<->] env M;
  ran_c:
    subst_ran G [-] subst_dom G [<->] env M
}.
```

Then the substitution on preterms (Definition 4.21) is defined as:

```
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.
```

Note that in the above definition we do not lift all the terms in the substitution upon encountering lambda abstraction but we count such abstractions and lift the term appropriately when it is being substituted. The motivation of this variant was performance as now the number of lifting operations in computing $t\gamma$ is proportional to the number of occurrences of variables in t that are in the domain of γ, whereas in the variant from Definition 4.21 it is the number of binders in t times the size of γ.

Finally typed substitution (Theorem 4.12):

```
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.
  (...)
Qed.
Definition subst (M: Term) (G: Subst) (C: correct_subst M G) : Term :=
    proj1_sig (subst_aux C).
```

4.4 Convertibility of Terms

During the development of computability (see Chapter 6) we needed to define computability for class of terms being equivalent from the point of view of computability. In other words we needed equivalence relation on terms $\sim$ such that:

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

(ii) it relates 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 to relate terms that differ only on names of free variables that is we want to have $t \sim u$ if there exist a renaming of variables γ such that $t = u\gamma$.

If $t \sim u$ then we will say that t and u are $\sim$-convertible.

We shortly present motivation for those three requirements:

(i) Typically we do not want to distinguish α-convertible terms in any way. This is also the easiest requirement as we are using de Bruijn indices to represent terms and in this representation α-convertible terms are simply equal.

(ii) The typical reasoning in computability proofs will be as follows: "given term $\Sigma \vdash t : \alpha \to \beta$ take variable $\Sigma \cdot \{x : \alpha\} \vdash x : \alpha$ and consider application $\Sigma \cdot \{x : \alpha\} \vdash @(t, x) : \beta \ldots$". Note that constructing such application requires extending Σ with the declaration for x. On the other hand we would like to have that the left argument of this application is equal to t. Strictly speaking it is not equal as it has an extended environment. But thanks to (ii) it will be $\sim$-convertible.

(iii) The reason behind this requirement is to have $\sim$-convertibility of lifted terms, so: $t \sim t{\uparrow}^i$ for any i. This in turn is needed for substitution. We will assume in Chapter 6 to have a substitution with all terms in its domain computable. But those terms are being substituted in context of some abstractions and hence need to be lifted (as explained in Section 4.3.2). So we want to be able to conclude computability of those lifted terms and since we are defining convertibility relation anyhow solving this problem by demanding that terms and their liftings are convertible seems to be rather natural.

We will spend the rest of this section seeking convertibility relation on terms $\sim$ satisfying posed requirements. The idea is, roughly speaking, to define two terms to be convertible if there exist an endomorphism on free variables of one term that maps them to free variables of the other term. We begin with the definition of such variable mappings.

Definition 4.24 (Variable mapping).
We say that a partial function $\Phi : \mathcal{V} \to \mathcal{V}$ is a variable mapping *if it is injective.*

Since Φ is injective function there exist 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 we can say when two environments or two preterms are convertible modulo this mapping.

Definition 4.25 (Environment convertibility).
Γ *and* Δ *are* convertible environments *modulo variable mapping Φ, denoted as: $\Gamma \overset{\Phi}{\approx} \Delta$, if:*

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

Definition 4.26 (Preterm convertibility).
We define t and t' to be convertible preterms *modulo variable mapping Φ, denoted as $t \overset{\Phi}{\approx} t'$*[2], *inductively as:*

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

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 of full environments then the following desired property does not hold:

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

To see that consider terms: $t = x : \alpha \vdash c : \delta$ and $u = x : \beta \vdash c : \delta$ and notice that we have $t \sim_{\emptyset} u$ but $t \not\sim_{\{(x,x)\}} u$ 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. The definition of term convertibility follows:

[2]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.

Definition 4.27 (Term convertibility, $\sim$).
Terms $\Gamma \vdash t : \alpha$ *and* $\Gamma' \vdash t' : \alpha'$ *are* convertible *up to 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 exist a variable mapping* Φ *such that* $\Gamma \vdash t : \alpha \overset{\Phi}{\sim} \Gamma' \vdash t' : \alpha'$.

Then we extend the notion of convertibility to substitutions.

Definition 4.28 (Convertible substitutions, $\sim$).
Substitutions γ *and* δ *are* convertible *with variable mapping* Φ*,* $\gamma \overset{\Phi}{\sim} \delta$*, iff:*

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

Theorem 4.13.
Relation $\sim$ *is an equivalence relation.*

Proof.

- **Reflexivity**. $t \overset{\Phi}{\sim} t$ with Φ being identity on variables restricted to free variables of t.
- **Symmetry**. If $t \overset{\Phi}{\sim} u$ then $u \overset{\Phi^{-1}}{\sim} t$.
- **Transitivity**. If $t \overset{\Phi}{\sim} u$ and $u \overset{\Psi}{\sim} w$ then $t \overset{\Phi \cdot \Psi}{\sim} w$.

□

The most important results concerning $\sim$ include:

- Compatibility with substitution: $t \sim t'$ and $\gamma \sim \gamma'$ implies $t\gamma \sim t'\gamma'$.
- Compatibility with beta-reduction, see Proposition 4.16.
- Computability with HORPO, see Proposition 7.4

Coq

The most interesting aspect of this part of the development 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 and then we will discuss it.

```
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 the domain and the codomain of `envSub` and `sizeOk` verifies that indeed that is the case. Finally `envSub_dec` states that `envSub` relation 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:

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

The convertibility of preterms can be then expressed as:

```
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.
```

Note the use of lifting of variable mappings in the case for lambda abstraction. For convertibility of environments we need only to look at free variables of terms (`activeEnv`) 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.
```

```
Notation "M ~ ( S ) N" := (terms_conv_with S M N) (at level 70).

Definition terms_conv M N := exists S, M ~(S) N.
Notation "M ~ N" := (terms_conv M N) (at level 70).
```

In Coq working on structures for which Leibniz equality does not denote the intended equality is not very easy. Setoid is an extension that makes it somehow 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. `terms_conv` was proven to be an equivalence relation (actual Coq proofs are somehow more complicated than what the proof of Theorem 4.13 would suggest) and registered as a setoid. Then we proved a number of operations to be morphisms with respect to `terms_conv`.

4.5 β-reduction

The β-reduction relation expresses the way in which application of a function to arguments is evaluated and hence is a model of computation for $\lambda^{\rightarrow}$.

Definition 4.29 ($\rightarrow_\beta$).
The β-reduction rule *is defined as:*

$$@(\lambda x\!:\!\alpha.t, u) \rightarrow_\beta t[x/u]$$

The β-reduction relation *is the smallest relation on terms that satisfied* β*-reduction rule and is closed under the following rules:*

$$\frac{t \rightarrow_\beta t'}{\lambda x\!:\!\alpha.t \rightarrow_\beta \lambda x\!:\!\alpha.t'}$$

$$\frac{t \rightarrow_\beta t'}{@(t,u) \rightarrow_\beta @(t',u)} \qquad \frac{u \rightarrow_\beta u'}{@(t,u) \rightarrow_\beta @(t,u')}$$

Proposition 4.14 (Subject reduction).
If $\Gamma \vdash t : \alpha \rightarrow_\beta \Gamma \vdash u : \beta$ *then* $\alpha = \beta$.

Proof. Induction on t. The only interesting case is a β-reduction step $@(\lambda x\!:\!\beta.t, u) \rightarrow_\beta t[x/u]$ but then $t\!:\!\alpha$ and $u\!:\!\beta$ and we conclude $t[x/u]\!:\!\alpha$ by Theorem 4.12. □

Proposition 4.15.
β-reduction preserves variables, that is:

$$t \rightarrow_\beta u \implies \mathsf{Vars}(t) \supseteq \mathsf{Vars}(u)$$

Proof. Induction on t. All cases but β-reduction step at the root easily by the induction hypothesis. For β-reduction step at the root $@(\lambda x : \beta.s, w) \rightarrow_\beta s[x/w]$ if x occurs in s then $\mathsf{Vars}(s[x/w]) = \mathsf{Vars}(s) \cup \mathsf{Vars}(w) = \mathsf{Vars}(@(\lambda x : \beta.s, w))$ otherwise $\mathsf{Vars}(s[x/w]) = \mathsf{Vars}(s) \subseteq \mathsf{Vars}(@(\lambda x : \beta.s, w))$. □

Proposition 4.16.
β-reduction is compatible with $\sim$, that is:

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

Proof. Induction on t. All cases but β-reduction step at the root easily by the induction hypothesis. For β-reduction step at the root $@(\lambda x : \beta.s, w) \to_\beta s[x/w]$ we get $t' = @(\lambda x : \beta.s', w')$ and $u' = s[x/w']$ with $s \overset{Q\uparrow^1}{\sim} s'$ and $w \overset{Q}{\sim} w'$ and hence $t' \to_\beta u'$. □

Proposition 4.17.
β-reduction is stable under substitution, that is:

$$t \to_\beta u \implies t\gamma \to_\beta u\gamma$$

Proof. Induction on t. All cases but β-reduction step at the root easily by the induction hypothesis. For β-reduction step at the root $@(\lambda x{:}\beta.s, w) \to_\beta s[x/w]$ since x does not occur free in w we get $t' = @(\lambda x{:}\beta.s, w)\gamma = @(\lambda x{:}\beta.s\gamma_{\setminus\{x\}}, w\gamma) \to_\beta s\gamma_{\setminus\{x\}}[x/w\gamma] = s[x/w]\gamma = u'$. □

We will need the following simple lemma in Chapter 7.

Lemma 4.18.

$$f(t_1, \ldots, t_n) \to_\beta u \implies \exists i \in \{1, \ldots, n\} \; . \; u = f(t_1, \ldots, t_i', \ldots, t_n) \land t_i \to_\beta t_i'$$

Proof. Follows easy from

$$@(t_1, \ldots, t_n) \to_\beta u \implies \exists i \in \{1, \ldots, n\} \; . \; u = @(t_1, \ldots, t_i', \ldots, t_n) \land t_i \to_\beta t_i'$$

that is easily proven by induction on n. □

Proposition 4.19.
β-reduction is monotonous, that is:

$$u \to_\beta u' \implies t[u]_p \to_\beta t[u']_p$$

Proof. Easy induction on p. □

Coq

The definition of β-reduction is done in two steps; first an arbitrary reduction compatible with term structure is defined. It is parameterized by a relation R. So a reduction is either a direct R-step at the root or a reduction in left or right argument of an application or a reduction in a body of an abstraction.

```
Inductive Reduction (R: relation Term) : Term -> Term -> Prop :=
| Step: forall M N,
    R M N ->
    Reduction R M N
| CompAppL: forall M N (Mapp: isApp M) (Napp: isApp N),
    Reduction R (appBodyL Mapp) (appBodyL Napp) ->
    appBodyR Mapp = appBodyR Napp ->
    Reduction R M N
| CompAppR: forall M N (Mapp: isApp M) (Napp: isApp N),
    Reduction R (appBodyR Mapp) (appBodyR Napp) ->
    appBodyL Mapp = appBodyL Napp ->
    Reduction R M N
| CompAbs: forall M N (Mabs: isAbs M) (Nabs: isAbs N),
    absType Mabs = absType Nabs ->
    Reduction R (absBody Mabs) (absBody Nabs) ->
    Reduction R M N.
```

Then a beta reduction step is defined using the substitution operation. Note the required lifting and lowering of terms.

```
Definition beta_subst:
  forall M (Mapp: isApp M) (MLabs: isAbs (appBodyL Mapp)),
    correct_subst (absBody MLabs) {x/(lift (appBodyR Mapp) 1)}.
Proof.
   (...)
Defined.

Inductive BetaStep : Term -> Term -> Prop :=
| Beta: forall M (Mapp: isApp M) (MLabs: isAbs (appBodyL Mapp)),
    BetaStep M (lower (subst (beta_subst M Mapp MLabs))
       (beta_lowering M Mapp MLabs)).

Definition BetaReduction := Reduction BetaStep.
Notation "M -b-> N" := (BetaReduction M N) (at level 30).
```

Chapter 5

Higher-Order Rewriting

In this Chapter we introduce the concept of higher-order rewriting, that is rewriting terms with bound variables. In Section 5.1 we introduce algebraic terms that we will use in Setion 5.2 where we present the AFS format for higher-order rewriting and shortly discuss its relation with other existing formats.

5.1 Terms

In this chapter we will introduce algebraic terms. The main difference with λ-terms as introduced in Section 4.1 is that now function symbols are algebraic operators equipped with arity. We use the definition of simple types $\mathcal{T}_{\mathcal{S}}$ over given set of sorts $\mathcal{S}$ from Section 4.1. Let us begin with the definition of algebraic signature.

Definition 5.1 (Algebraic signature).
A declaration of a function symbol f expecting n arguments of types $\alpha_1, \ldots, \alpha_n$ and an output type β we will write as $f : \alpha_1 \times \ldots \times \alpha_n \to \beta$.

An algebraic signature *$\mathcal{F}$ is a set of such function declarations.*

Now we define algebraic terms as follows:

Definition 5.2 (Algebraic terms).
A set of algebraic terms *is defined by the following grammar:*

$$\mathcal{P}t_a := \mathcal{V} \mid @(\mathcal{P}t_a, \mathcal{P}t_a) \mid \lambda\mathcal{V}\!:\!\mathcal{T}_{\mathcal{S}}.\mathcal{P}t_a \mid \mathcal{F}(\mathcal{P}t_a, \ldots, \mathcal{P}t_a)$$

and such terms conform to the following typing rules (compare with Definition 4.5):

$$\frac{x\!:\!\alpha \in \Gamma}{\Gamma \vdash x : \alpha} \qquad \frac{\begin{array}{c} f : \alpha_1 \times \ldots \times \alpha_n \to \beta \in \Sigma \\ \Gamma \vdash t_1 : \alpha_1, \ldots, \Gamma \vdash t_n : \alpha_n \end{array}}{\Gamma \vdash f(t_1, \ldots, t_n) : \beta}$$

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

Example 5.1.
Consider two sorts: $\mathbb{N}$ for natural numbers and List representing lists of natural numbers.

$$\mathcal{S} = \{\mathbb{N}, \mathsf{List}\}$$

Now consider the following signature $\mathcal{F}$:

$$\begin{aligned} \mathcal{F} = \{&\mathsf{nil} : \mathsf{List}, \\ &\mathsf{cons} : \mathbb{N} \times \mathsf{List} \to \mathsf{List}, \\ &\mathsf{map} : \mathsf{List} \times (\mathbb{N} \to \mathbb{N}) \to \mathsf{List}\} \end{aligned}$$

Some terms over this signature:

$$\emptyset \vdash \mathsf{nil} : \mathsf{List}$$
$$X : \mathbb{N} \to \mathbb{N} \vdash \mathsf{map}(\mathsf{nil}, X) : \mathsf{List}$$
$$x : \mathbb{N}, l : \mathsf{List}, X : \mathbb{N} \to \mathbb{N} \vdash \mathsf{map}(\mathsf{cons}(x, l), X) : \mathsf{List}$$
$$x : \mathbb{N}, l : \mathsf{List}, X : \mathbb{N} \to \mathbb{N} \vdash \mathsf{cons}(@(X, x), \mathsf{map}(l, X)) : \mathsf{List}$$

Note that in the formalization our intention is to use $\lambda^{\to}$ terms to represent such algebraic terms. To avoid dealing with arities we made a simplification and assumed that output types of functions are base types, an assumption often made in the literature. This allows us to represent a function $f : \alpha_1 \times \ldots \times \alpha_n \to \beta$ by a $\lambda^{\to}$ constant $f : \alpha_1 \to \ldots \to \alpha_n \to \beta$ and its application $f(t_1, \ldots, t_n)$ as a $\lambda^{\to}$ term $@(f, t_1, \ldots, t_n)$.

Remark. All the theory in the following chapters deals with algebraic terms. They are assumed to be encoded in this way and hence all the definitions of substitution, positions etc. from Chapter 4 need not be repeated here for this new algebraic structure.

Coq

As remarked above the higher-order terms with arity were encoded in Coq using simply typed lambda-terms. However to avoid introducing arity we assumed that the output type of every function is a simple type and hence a function $f : \alpha_1 \times \ldots \times \alpha_n \to \beta$ can be represented as a fully applied lambda term $@(f, t_1, \ldots, t_n)$.

So effectively algebraic terms are a subset of simply typed lambda terms where every function is fully applied. Please note that this puts no restriction on types and/or level of application for variables. This condition has been formalized by the following predicate:

```
Inductive algebraic: Term -> Prop :=
| AlgVar: forall M,
     isVar M ->
     algebraic M
| AlgAbs: forall M (Mabs: isAbs M),
     algebraic (absBody Mabs) ->
     algebraic M
| AlgApp: forall M,
     isApp M ->
     ~isFunApp M ->
     (forall M’, isAppUnit M’ M -> algebraic M’) ->
     algebraic M
| AlgFunApp: forall M,
     isFunApp M ->
     isBaseType (type M) ->
     (forall M’, isArg M’ M -> algebraic M’) ->
     algebraic M.
```

Now every time an algebraic term is expected we take a pair of a lambda term (`T: Term`) and a proof that it satisfies this condition (`Talg: algebraic T`). A neater solution would be to introduce a type for algebraic terms as a refinement of lambda terms and then introduce a coercion between the two.

```
Definition ATerm := { T: Term | algebraic T }.
Definition aterm2term (A: ATerm) : Term := proj1_sig A.
Coercion aterm2term : ATerm >-> Term.
```

5.2 Rewriting

There are several variants of higher-order rewriting. Here we use the algebraic-functional systems (AFSs) introduced by Jouannaud and Okada [18]. The main difference between AFSs and another popular format of higher-order rewriting systems (HRSs, [27]) is that in HRSs we work modulo beta-eta (using pure $\lambda^{\rightarrow}$ terms) whereas in AFSs we do not (and function symbols have fixed arity, as in Definition 5.2). As a consequence rewriting for AFSs is defined using plain pattern matching compared to rewriting modulo $\beta\eta$ of $\lambda^{\rightarrow}$ in HRSs framework. For a broader discussion on this subject we refer the reader to, for instance, [31]. The presentation in this section follows [20].

We give definitions of higher-order: rewrite rules, rewriting systems and rewrite relation.

Definition 5.3 (Higher-order term rewriting system).
Given a signature $\mathcal{F}$ a rewrite rule *is a quadruple $\Gamma \vdash \ell \rightarrow r : \alpha$ where ℓ and r are algebraic terms such that:*

- $\mathsf{Vars}(r) \subseteq \mathsf{Vars}(l)$
- $\Gamma \vdash \ell : \alpha$ *and* $\Gamma \vdash r : \alpha$.

A higher-order rewriting system *is a set of rewrite rules.*

Definition 5.4 (The rewrite relation).
Given higher-order rewriting system R a term $\Gamma \vdash s : \alpha$ rewrites to a term $\Gamma \vdash t : \alpha$ if there exist a rewrite rule $\Delta \vdash \ell \rightarrow r : \beta \in R$, a substitution γ and a position p such that:

- $\mathit{Dom}(\gamma) \subseteq \Delta$,
- $\Delta \cdot \mathit{Ran}(\gamma) \subseteq \Gamma_{s|_p}$,
- $s_{|_p} = \ell\gamma$,
- $t = s[r\gamma]_p$.

Example 5.2 (Example 5.1 continued).
With signature given in Example 5.1 we can construct the following higher-order term rewriting system (from [19]) representing the usual map computation on lists of natural numbers:

$$\begin{array}{rcrcl} X:\mathbb{N}\rightarrow\mathbb{N} & \vdash & \mathsf{map}(\mathsf{nil}, X) & \rightarrow & \mathsf{nil} \\ x:\mathbb{N}, l:\mathsf{List}, X:\mathbb{N}\rightarrow\mathbb{N} & \vdash & \mathsf{map}(\mathsf{cons}(x,l), X) & \rightarrow & \mathsf{cons}(@(X,x), \mathsf{map}(l,X)) \end{array}$$

For more detailed introduction to higher-order rewriting in AFS format and proving its termination by means of higher-order reduction orderings we refer the reader to [20].

Coq

Note that Definitions 5.3 and 5.4 do not play any direct role for our results and hence are omitted in the formalization.

Chapter 6

Computability

In this chapter we present the computability predicate proof method due to Tait and Girard [17, 34]. In Chapter 7.3 we will use computability with respect to a particular relation (being union of HORPO and β-reduction relation in that case) but we present computability for an arbitrary relation satisfying given properties.

We begin by giving a definition of computability in Section 6.1 and in Section 6.2 we prove some computability properties that we will need in Chapter 7.

6.1 Definition of Computability

We begin by presenting the definition of computability predicate.

Definition 6.1 (Computability).
A term $t : \delta$ is computable with respect to a relation on terms $\gg$, denoted as $t \in \mathbb{C}_\delta$ (or simply $t \in \mathbb{C}$ if type is of no interest), if:

- *δ is a base type and t is strongly normalizable with respect to $\gg$ ($t \in \mathcal{A}cc_{\gg}$), or*
- *$\delta = \alpha \to \beta$ and $@(t', u) \in \mathbb{C}_\beta$ for every $t' \sim t$ and all $u \in \mathbb{C}_\alpha$.*

This definition deserves few words of explanation. Firstly, it is usual to assume that variables are computable. We do not do that, following the presentation in [19] and we prove that variables are computable as one of the computability properties.

Another deviation from the standard definition (as in [17]) is the fact that we define computability modulo convertibility relation on terms ($\sim$). That is because a typical pattern in computability proof will be as follows: "for $t \in \mathbb{C}_{\alpha\to\beta}$ take variable a fresh $x : \alpha$ and consider $@(t, x) : \beta$ having $@(t, x) \in \mathbb{C}_\beta$ from the definition of computability". But constructing the application $@(t, x)$ requires extending environment of t with a declaration for x. Such subtleties are usually omitted in a presentation but our goal is to make presentation that closely reflects the formal verification that has been made. That is why we define computability modulo $\sim$, which will also prove helpful for dealing with computability of lifted terms as we shall see later.

Coq

The coding of the definition of computability in Coq is somehow tricky. The problem is that it needs to be expressed as a fixpoint definition and Coq uses a simple criterion to ensure that such definitions are terminating, namely one of the arguments in the recursive call needs to be a subterm of the original argument. This is not the case for computability. To check whether $t : \alpha \to \beta$ is computable we check whether its application to a computable term $u : \alpha$ is computable.

Although types of $u : \alpha$ and $@(t, u) : \beta$ are simpler than of $t : \alpha \to \beta$ this is not enough for a simple syntactic criterion of Coq. What makes matters even worse is that actually we do not take $@(t, u) : \beta$ but $@(t', u) : \beta$ with $t \sim t'$. Hence we extracted the type of a term as an extra argument to an auxiliary `ComputableS` function. This argument satisfies Coq requirements of decreasing arguments and hence Coq accepts this definition. Then `Computable` merely calls `ComputableS` with the appropriate type.

```
Fixpoint ComputableS (M: Term) (T: SimpleType) {struct T} : Prop :=
  algebraic M /\
  type M = T /\
  match T with
  | #T => AccR M
  | TL --> TR =>
    forall P (Papp: isApp P) (PL: appBodyL Papp ~ M)
      (typeL: type P = TR) (typeR: type (appBodyR Papp) = TL),
      algebraic (appBodyR Papp) ->
      ComputableS (appBodyR Papp) TL ->
      ComputableS P TR
  end.

Definition Computable M := ComputableS M (type M).
```

It is worth noting that a new `Function` feature of Coq 8.1, allowing for more complex fixpoint definitions where the obligation of proving that some argument is decreasing is left to the user, could be very helpful in this and many other similar situations. However at the time of writing this article only Coq 8.1beta is available and in this version this feature is not powerful enough to deal with our variant of computability.

6.2 Computability Properties

We want to abstract away from the particular relation with respect to which we define computability. But in order to prove required computability properties we need to make some assumptions about this relation. Figure 6.1 presents the list of properties we require $\gg$ to conform to. All those properties but (P_8) are quite general and natural so our abstraction is (partly) successful. The property (P_8) looks rather complicated but basically it states that every reduction of an application either operates on separate arguments or it is a β-reduction step. This property is rather specific for a particular $\gg$ relation being in that case union of β-reduction and HORPO as we will use it in Chapter 7.

Figure 6.2 on the other hand presents the list of all the computability properties that we will need in the following chapter. We proceed with presenting proofs for those properties. We begin with two simple auxiliary lemmas.

Lemma 6.1.
Let $@(t, u) \in \mathcal{A}cc$ then $t \in \mathcal{A}cc$.

Proof. Easy using monotonicity (P_7). □

Lemma 6.2.
Let $t \in \mathcal{A}cc$ and $t \sim u$ then $u \in \mathcal{A}cc$.

Proof. Easy using compatibility with $\sim$ (P_5). □

Let us recall that we did not assume variables to be computable. Variables of a base type are computable due to the definition of computability and the assumption that variables are not

Figure 6.1: Abstract properties assumed for $\gg$.

(P_1)	Subject reduction $t : \alpha \gg u : \beta \implies \alpha = \beta$
(P_2)	Preservation of environments $\Gamma_t \vdash t : \delta \gg \Gamma_u \vdash u : \eta \implies \Gamma_t = \Gamma_u$
(P_3)	Preservation of variables $t \gg u \implies \mathsf{Vars}(t) \supseteq \mathsf{Vars}(u)$
(P_4)	Normal form of variables $\neg(x \gg u)$
(P_5)	Compatibility with $\sim$ $\left.\begin{array}{c} \Gamma \vdash t : \delta \gg \Delta \vdash u : \eta \\ \Gamma \vdash t : \delta \overset{Q}{\sim} \Gamma' \vdash t' : \delta' \\ \Delta \vdash u : \eta \overset{Q}{\sim} \Delta' \vdash u' : \eta' \\ \Gamma' = \Delta' \end{array}\right\} \Longrightarrow \Gamma' \vdash t' : \delta' \gg \Delta' \vdash u' : \eta'$
(P_6)	Stability under substitution $t \gg u \implies t\gamma \gg u\gamma$
(P_7)	Monotonicity $u \gg u' \implies t[u]_p \gg t[u']_p$
(P_8)	Reductions of applications $t = @(t_l, t_r) \gg u \implies$ $\left\{\begin{array}{c} \exists t_b \ . \ t = @(\lambda x\!:\!\alpha.t_b, t_r) \wedge u = t_b[x/t_r] \\ \vee \\ \exists u_l, u_r \ . \ u = @(u_l, u_r) \wedge \left(\begin{array}{c} t_l = u_l \wedge t_r \gg u_r \\ \vee \\ t_l \gg u_l \wedge t_r = u_r \\ \vee \\ t_l \gg u_l \wedge t_r \gg u_r \end{array}\right) \end{array}\right.$
(P_9)	Reductions of abstraction $\lambda x\!:\!\alpha.t_b \gg u \implies \exists u_b \ . \ u = \lambda x\!:\!\alpha.u_b \wedge t_b \gg u_b$

Figure 6.2: Computability properties.

(C_1)	Every computable term is strongly normalizable. $t \in \mathbb{C}_\delta \implies t \in \mathcal{A}cc$ Lemma 6.3
(C_2)	Every reduct of a computable term is computable. $t \in \mathbb{C}_\delta \wedge t \gg u \implies u \in \mathbb{C}_\delta$ Lemma 6.3
(C_3)	Neutral term is computable iff its every reduct is computable. t-neutral $\implies ((\forall u \,.\, t \gg u \implies u \in \mathbb{C}_\delta) \iff t \in \mathbb{C}_\delta)$ Lemma 6.3
(C_4)	Variables are computable. $\forall x : \delta \,.\, x \in \mathbb{C}_\delta$ Lemma 6.4
(C_5)	Computability of abstractions $\forall u \in \mathbb{C}_\alpha \,.\, t[x/u] \in \mathbb{C}_\beta \implies (\lambda x\!:\!\alpha.t) \in \mathbb{C}_{\alpha\to\beta}$ Lemma 6.5
(C_6)	Term convertible with computable term is computable. $t \in \mathbb{C}_\delta \wedge t \sim t' \implies t' \in \mathbb{C}_\delta$ Lemma 6.6

reducible (P_4). Variables of a functional type are computable by property (C_3) presented in the following lemma which forbids us to prove the following computability properties (C_1), (C_2) and (C_3) separately.

Lemma 6.3.
For all terms $\Gamma \vdash t : \delta$, $\Delta \vdash u : \delta$ *we prove that:*

(C_1) $t \in \mathbb{C}_\delta \implies t \in \mathcal{A}cc$

(C_2) $t \in \mathbb{C}_\delta \wedge t \gg u \implies u \in \mathbb{C}_\delta$

(C_3) *if t-neutral then* $(\forall w\!:\!\delta \,.\, t \gg w \implies w \in \mathbb{C}_\delta) \iff t \in \mathbb{C}_\delta$

Proof. Induction on type δ. Note that 'if' part of (C_3) is (C_2) so below we only prove the 'only if' part of this property.

- δ is a base type.
  - (C_1) $t \in \mathbb{C}_\delta$ and δ is a base type so $t \in \mathcal{A}cc$ by the definition of computability.
  - (C_2) $t \in \mathcal{A}cc$ by the same argument as in case for (C_1). $t \in \mathcal{A}cc$ and $t \gg u$ hence $u \in \mathcal{A}cc$. By subject reduction for $\gg$ (P_1), $u\!:\!\delta$, so $u \in \mathbb{C}_\delta$ by the definition of computability.
  - (C_3) $t\!:\!\delta$ so to show $t \in \mathbb{C}_\delta$ we need to show $t \in \mathcal{A}cc$ but for every w such that $t \gg w$ we have $w \in \mathbb{C}_\delta$ by assumption. Hence $w \in \mathcal{A}cc$ by the definition of computability and $t \in \mathcal{A}cc$.
- $\delta = \alpha \to \beta$

(C_1) Take variable $x:\alpha$ which is computable by induction hypothesis (C_3) as variables are not reducible by (P_4). Now consider application $\Gamma \cup \{x:\alpha\} \vdash @(t,x) : \beta$ which is computable by the definition of computability (note that $\Gamma \cup \{x : \alpha\} \vdash t : \delta \sim \Gamma \vdash t : \delta$). So $\Gamma \cup \{x:\alpha\} \vdash @(t,x) : \beta \in \mathcal{A}cc$ by induction hypothesis (i). Then $\Gamma \cup \{x:\alpha\} \vdash t : \delta \in \mathcal{A}cc$ by Lemma 6.1 and $\Gamma \vdash t : \delta \in \mathcal{A}cc$ by Lemma 6.2.

Remark. From now on we will work modulo $\sim$ without stating it explicitly which greatly improves the readability of the proofs. The reader interested in all the details is encouraged to consult the Coq scripts.

(C_2) By the definition of computability $u \in \mathbb{C}_{\alpha\to\beta}$ if for every $s \in \mathbb{C}_\alpha$, $@(u,s) \in \mathbb{C}_\beta$. $@(t,s) \in \mathbb{C}_\beta$ by the definition of computability and $@(t,s) \gg @(u,s)$ by monotonicity assumption (P_7). Finally we conclude $@(u,s) \in \mathbb{C}_\beta$ by the induction hypothesis (C_2).

(C_3) By the definition of computability $t \in \mathbb{C}_{\alpha\to\beta}$ if for every $s \in \mathbb{C}_\alpha$, $@(t,s) \in \mathbb{C}_\beta$. By induction hypothesis for (C_1), $s \in \mathcal{A}cc$ so we continue by well-founded inner induction on s with respect to $\gg$.

$@(t,s) : \beta$ is neutral so we can apply induction hypothesis for (C_3) and we are left to show that all reducts of $@(t,s)$ are computable. We do case analysis using (P_9). Since t is neutral and hence is not an abstraction, we can exclude the β-reduction case and we are left with the following cases:

- $@(t,s) \gg @(t',s)$ with $t \gg t'$. Then t' is computable as so is every reduct of t and application of two computable terms is computable by the definition of computability.
- $@(t,s) \gg @(t,s')$ with $s \gg s'$. We observe that $s' \in \mathbb{C}$ by induction hypothesis for (C_2) and since $s \gg s'$ we apply the inner induction hypothesis to conclude $(t,s') \in \mathbb{C}_\beta$.
- $@(t,s) \gg @(t',s')$ with $t \gg t'$ and $s \gg s'$. Every reduct of t is computable so $t' \in \mathbb{C}_{\alpha\to\beta}$. By the induction hypothesis for (C_2) $s' \in \mathbb{C}_\alpha$. Again application of two computable terms is computable.

□

An easy consequence of the above lemma is the fact that all variables are computable.

Lemma 6.4 (C_4).
For every variable $x:\delta$, $x \in \mathbb{C}_\delta$.

Proof. Variables are neutral so we apply (C_3) and since variables are in normal forms due to (P_4) we conclude $x \in \mathbb{C}_\delta$. □

The following property deals with computability of abstractions.

Lemma 6.5 (C_5).
Consider abstraction $(\lambda x:\alpha.t) : \alpha \to \beta$. If for every $u \in \mathbb{C}_\alpha$, $t[x/u] \in \mathbb{C}_\beta$ then $(\lambda x:\alpha.t) \in \mathbb{C}_{\alpha\to\beta}$.

Proof. By the definition of computability $\lambda x:\alpha.t$ is computable if for every $s \in \mathbb{C}_\alpha$, $@(\lambda x:\alpha.t, s) \in \mathbb{C}_\beta$. Note that $t \in \mathbb{C}$ by assumption because $t = t[x/x]$ and $x \in \mathbb{C}$ by (C_4). So by (C_1) both $t \in \mathcal{A}cc$ and $s \in \mathcal{A}cc$ and we proceed by well-founded part induction on a pair of computable terms (t,s) with respect to ordering $\rhd = (\gg,\gg)_{lex}$. Now, since $@(\lambda x:\alpha.t, s)$ is neutral, by (C_3) we are left to show that all its reducts are computable. Let us continue by considering possible reducts of this application using (P_9). So we have $@(\lambda x:\alpha.t, s) \gg u$ and the following cases to consider:

- $u = t[x/s]$. $u \in \mathbb{C}$ by the assumption.
- $u = @(\lambda x:\alpha.t, s')$ with $s \gg s'$. $u \in \mathbb{C}$ by the induction hypothesis for $(t,s') \lhd (t,s)$.

- $u = @(w, s)$ with $\lambda x:\alpha.t \gg w$. By (P_8) we know that this reduction is in the abstraction body of $\lambda x:\alpha.t$ so in fact $w = \lambda x:\alpha.t'$ with $t \gg t'$. We conclude computability of u by the induction hypothesis for $(t', s) \lhd (t, s)$.
- $u = @(w, s')$ with $\lambda x:\alpha.t \gg w$ and $s \gg s'$. As in the above case, by (P_8) we observe that $w = \lambda x:\alpha.t'$ with $t \gg t'$ and we conclude computability of u by the induction hypothesis for $(t', s') \lhd (t, s)$.

□

We conclude with the following simple property.

Lemma 6.6 (C_6)**.**
If $t \in \mathbb{C}_\delta$ and $t \sim t'$ then $t' \in \mathbb{C}_\delta$

Proof. If δ is a simple type then we apply Lemma 6.2. If δ is an arrow type then we conclude $t' \in \mathbb{C}_\delta$ directly from the definition of computability for t. Note that here we make use of the fact that we defined computability modulo $\sim$. □

Coq

Proving computability properties turned out to be the most difficult part of the whole development. In the first version of the development ([23]) those properties were assumed as axioms. Completing the pursuit of making the development axiom-free and proving all computability properties turned out to be a very laborious task after which the size of Coq script tripled.

Strictly speaking in terms of script size, the part of the formalization dealing with computability accounts for only slightly more than 5%. However, as those properties are at the heart of proofs concerning HORPO relation, providing proofs for them triggered many other developments.

This difficulty can be partially explained by the real complexity of the computability predicate proof method. Other factors that contributed to making this task difficult include:

- the fact that algebraic terms were encoded using pure $\lambda^{\rightarrow}$ terms,
- the necessity of defining computability modulo $\sim$.

For the clarity of presentation those issues are left implicit in the computability proofs presented in this section but in Coq proofs they had to be taken care of. Another aspect not visible in this presentation is the use of de Bruijn indices [11] to represent terms.

Chapter 7

Higher-Order Recursive Path Ordering (HORPO)

In this chapter we present the core of this work: the results concerning the higher-order recursive path ordering (HORPO). We begin by presenting the definition of HORPO in Section 7.1, then some of its properties in Section 7.2 and its main property – well-foundedness – in Section 7.3.

The material of this chapter will appear in [24].

7.1 Definition of the Ordering

As indicated in the introduction, Chapter 1, the subject of our formalization is a slight variant of the HORPO ordering as presented in [19]. We begin by first presenting the formalized variant of the definition and then we discuss the differences comparing to the original definition of Jouannaud and Rubio.

Definition 7.1 (The higher-order recursive path ordering, $\succ$).
Assume a well-founded order on the set of function symbols $\rhd$, called a precedence. We define HORPO *ordering $\succ$ on terms and in this definition by $\succeq$ we denote reflexive closure of* HORPO *(that is $\succeq \; \equiv \; \succ \cup =$) and by $\succ_{MUL}$ its multiset extension.*

$\Gamma \vdash t : \delta \succ \Gamma \vdash u : \delta$ *iff one of the following holds:*

(H_1) $t = f(t_1, \dots, t_n)$
$\exists i \in \{1, \dots, n\} \; . \; t_i \succeq u$

(H_2) $t = f(t_1, \dots, t_n)$, $u = g(u_1, \dots, u_k)$
$f \rhd g$
$t \succ\!\!\succ \{u_1, \dots u_k\}$

(H_3) $t = f(t_1, \dots, t_n)$, $u = f(u_1, \dots, u_k)$
$\{\{t_1, \dots t_n\}\} \succ_{MUL} \{\{u_1, \dots, u_k\}\}$

(H_4) $@(u_1, \dots, u_k)$ *is a partial flattening of* u
$t \succ\!\!\succ \{u_1, \dots u_k\}$

(H_5) $t = @(t_l, u_r)$, $u = @(t_l, u_r)$
$\{\{t_l, t_r\}\} \succ_{MUL} \{\{u_l, u_r\}\}$

(H_6) $t = \lambda x\!:\!\alpha.t'$, $u = \lambda x\!:\!\alpha.u'$
$t' \succ u'$

where $\succ\!\!\succ$ *is a relation between a term and a set of terms, defined as:*
$t = f(t_1, \ldots, t_k) \succ\!\!\succ \{u_1, \ldots, u_n\}$ *iff* $\forall i \in \{1, \ldots, n\} \,.\, t \succ u_i \vee (\exists j \,.\, t_j \succeq u_i)$.

Note that, following Jouannaud and Rubio, we do not prove HORPO to be an ordering. In the following sections we will prove its well-foundedness and thus its transitive closure will be a well-founded ordering. There are three major differences between our definition and the definition from [19].

First let us note that in our variant only terms of equal types can be compared whereas in the original definition this restriction is weaker and it is possible to compare terms of equivalent types, where equivalence of types is a congruence generated by equating all sorts (in other words two types are equivalent if they have the same arrow structure). The reason for strengthening this assumption is that allowing to reduce between different sorts poses some technical difficulties. In [19] this problem was solved by extending the typing rules with the congruence rule which presence is basically equivalent to collapsing all sorts and which allows typing terms that normally would be ill-typed due to a sort clash. Our goal was to use $\lambda^{\rightarrow}$ in its purest form as a meta-language and hence we decided not to do that. Note however that this remark is relevant only for many-sorted signatures as for one sorted signatures the notions of type equality and type equivalence coincide.

Second difference is that the original definition of HORPO uses statues and allows arguments of function symbols to be compared either lexicographically or as multisets, depending on the status, whereas we allow only for comparing arguments of functions as multisets. This choice was made simply to avoid dealing with statuses and multiset comparison has been chosen as posing more difficulties, so extension with statuses and lexicographic comparison should be relatively easy.

Finally we use the multiset ordering as in Definition 3.3 instead of the one from Definition 3.2. Case (ii) of Proposition 3.4 will be crucial for the results in Section 7.2 and for the Definition 3.2 only its weaker variant holds (Proposition 3.4 (i)).

We conclude this section with a simple termination argument using HORPO.

Example 7.1 (Example 5.2 continued).
Consider the higher-order term rewriting system from Example 5.2. We will try to orient the rules of this system using HORPO. *The first one is trivial by* (H_1). *For the second one we take precedence with* map $>$ cons *and apply* (H_2). *The remaining obligations are* $\mathsf{map}(\mathsf{cons}(x, l), X) \succ @(X, x)$ *and* $\mathsf{map}(\mathsf{cons}(x, l), X) \succ \mathsf{map}(l, X)$. *The latter is easily shown by* (H_3) *and* (H_1). *For the first inequality we would like to apply* (H_5) *but the problem is that we have a sort clash: the first term has type* List *whereas the second one* $\mathbb{N}$. *This is the place where the difference between our variant of* HORPO *with the definition from [19] shows up. In this case it prevents us from orienting the second rule of this system. However if we consider the variant of this system with unified sorts for* $\mathbb{N}$ *and* List *then we have* $\mathsf{map}(\mathsf{cons}(x, l), X) \succ @(X, x)$ *by* (H_5) *followed by* (H_1) *and both rules can be oriented.*

Coq

We will shortly discuss the way in which Definition 7.1 is expressed in Coq. We begin with some shorthands for multisets and lists of terms and a notation for precedence.

```
Definition TermMul := Multiset.
Definition TermList := list Term.
Notation "f >#> g" := (Prec.P.O.gtA f g) (at level 30).
```

Then the main definition contains 5 mutually recursive Coq definitions.

Notation	Coq notation	Coq definition	Description
$\succ\!\succ$	`[>>]`	`horpoArgs`	Auxiliary condition in the definition of HORPO.
	`>->`	`prehorpo`	HORPO clauses with restrictions on equal types and environments left out.
$\succ$	`>>`	`horpo`	HORPO ordering.
$\succ_{MUL}$	`{>>}`	`horpoMul`	Multiset extension of HORPO.
	`>>=`	`horpoRC`	Reflexive closure of HORPO.

The code of those mutually inductive definitions follows. Although it is long, it is merely a straightforward coding of Definition 7.1.

```
Inductive horpoArgs : Term -> TermList -> Prop :=
| HArgsNil: forall M, M [>>] nil
| HArgsConsEqT: forall M N TL, M >> N -> M [>>] TL -> M [>>] (N :: TL)
| HArgsConsNotEqT: forall M N TL, (exists2 M', isArg M' M & M' >>= N) ->
    M [>>] TL -> M [>>] (N :: TL)
where
   "M [>>] N" := (horpoArgs M N)

with prehorpo : Term -> Term -> Prop :=
| HSub: forall M N, isFunApp M -> (exists2 M', isArg M' M & M' >>= N) -> M >-> N
| HFun: forall M N f g, term (appHead M) = ^f -> term (appHead N) = ^g ->
    f >#> g -> M [>>] (appArgs N) -> M >-> N
| HMul: forall M N f, term (appHead M) = ^f -> term (appHead N) = ^f ->
    list2multiset (appArgs M) {>>} list2multiset (appArgs N) -> M >-> N
| HAppFlat: forall M N Ns, isFunApp M -> isPartialFlattening Ns N -> M [>>] Ns ->
    M >-> N
| HApp: forall M N (MApp: isApp M) (NApp: isApp N), ~isFunApp M ->
    {{ appBodyL MApp, appBodyR MApp }} {>>} {{ appBodyL NApp, appBodyR NApp }} ->
    M >-> N
| HAbs: forall M N (MAbs: isAbs M) (NAbs: isAbs N),
    absBody MAbs >> absBody NAbs -> M >-> N
where
   "M >-> N" := (prehorpo M N)

with horpo : Term -> Term -> Prop :=
| Horpo: forall M N, type M = type N -> env M = env N -> algebraic M ->
           algebraic N -> M >-> N -> M >> N
where
   "M >> N" := (horpo M N)

with horpoMul : TermMul -> TermMul -> Prop :=
| HMulOrd: forall (M N: TermMul), MSetOrd.MultisetGT horpo M N -> M {>>} N
where
   "M {>>} N" := (horpoMul M N)

with horpoRC : Term -> Term -> Prop :=
| horpoRC_step: forall M N, M >> N -> M >>= N
| horpoRC_refl: forall M, M >>= M
where
   "M >>= N" := (horpoRC M N).
```

7.2 Properties of the Ordering

In this section we will prove some properties of HORPO.

Proposition 7.1.
HORPO *is stable under substitution, that is:*

$$t \succ u \implies t\gamma \succ u\gamma$$

Proof. Induction on pair (t, u) ordered by $(\sqsubset, \sqsubset)_{lex}$ followed by a case analysis on $t \succ u$.

(H_1) $t = f(t_1, \ldots, t_n)$ and $t_i \succeq u$ for some $i \in \{1, \ldots, n\}$. But then $t\gamma = f(t_1\gamma, \ldots, t_n\gamma) \succ u\gamma$ by (H_1) since $t_i\gamma \succeq u\gamma$ by the induction hypothesis.

(H_2) $t = f(t_1, \ldots, t_n)$, $u = g(u_1, \ldots, u_k)$, $f \rhd g$ and $t \succ\!\!\succ \{u_1, \ldots, u_k\}$. But then to get $t\gamma \succ u\gamma$ by (H_2) we only need to show $t\gamma \succ\!\!\succ \{u_1\gamma, \ldots, u_k\gamma\}$. For every $i \in \{1, \ldots, k\}$ we have $t \succ u_i \lor (\exists j \;.\; t_j \succeq u_i)$. In either case we have $t\gamma \succ u_i\gamma$ or $t_j\gamma \succeq u_i\gamma$ by the induction hypothesis.

(H_3) $t = f(t_1, \ldots, t_n)$, $u = f(u_1, \ldots, u_k)$ and $\{\{t_1, \ldots t_n\}\} \succ_{MUL} \{\{u_1, \ldots, u_k\}\}$. But then we have $\{\{t_1\gamma, \ldots t_n\gamma\}\} \succ_{MUL} \{\{u_1\gamma, \ldots, u_k\gamma\}\}$ since for all $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, k\}$, $t_i \succ u_j$ implies $t_i\gamma \succ u_j\gamma$ by the induction hypothesis. So we get $t\gamma \succ u\gamma$ by (H_3).

(H_4) $@(u_1, \ldots, u_k)$ is a partial flattening of u and $t \succ\!\!\succ \{u_1, \ldots u_k\}$. We use the same partial flattening for $u\gamma$ and get $t\gamma \succ\!\!\succ \{u_1\gamma, \ldots, u_k\gamma\}$ with the same argument as in case (H_2). We conclude $t\gamma \succ u\gamma$ by (H_4).

(H_5) $t = @(t_l, u_r)$, $u = @(t_l, u_r)$ and $\{\{t_l, t_r\}\} \succ_{MUL} \{\{u_l, u_r\}\}$. Type considerations show that $t_l \succeq u_l$, $t_r \succeq u_r$ and $t_l \succ u_l \lor t_r \succ u_r$. By induction hypothesis on $(t_l\gamma, u_l\gamma)$ and $(t_r\gamma, u_r\gamma)$ we conclude $\{\{t_l\gamma, t_r\gamma\}\} \succ_{MUL} \{\{u_l\gamma, u_r\gamma\}\}$ and hence $t\gamma \succ u\gamma$ by (H_5).

(H_6) $t = \lambda x\!:\!\alpha.t'$, $u = \lambda x\!:\!\alpha.u'$ and $t' \succ u'$. But then $t\gamma = \lambda x\!:\!\alpha.t'\gamma$, $u\gamma = \lambda x\!:\!\alpha.u'\gamma$ and $t'\gamma \succ u'\gamma$ by the induction hypothesis. So $t\gamma \succ u\gamma$ by (H_6).

□

Proposition 7.2.
HORPO *is monotonous, that is:*

$$u \succ u' \implies t[u]_p \succ t[p]_{u'}$$

Proof. The proof proceeds by induction on p and essentially uses the following observations:

- if $w_r \succ w_r'$ then $@(w_l, w_r) \succ @(w_l, w_r')$ by (H_5).
- if $w_l \succ w_l'$ then $@(w_l, w_r) \succ @(w_l', w_r)$ by (H_5).
- if $w \succ w'$ then $f(\ldots, w, \ldots) \succ f(\ldots, w', \ldots)$ by (H_3).
- if $w \succ w'$ then $\lambda x\!:\!\alpha.w \succ \lambda x\!:\!\alpha.w'$ by (H_6).

□

Proposition 7.3.
HORPO *preserves variables, that is:*

$$t \succ u \implies \mathsf{Vars}(t) \supseteq \mathsf{Vars}(u)$$

Proof. The proof uses the same inductive argument as in the above proof of stability of HORPO under substitution and all cases are easy. □

Proposition 7.4.
HORPO is compatible with $\sim$, that is:

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

Proof. The proof is slightly technical but the inductive argument is again the same: induction on pair of terms (t, u) ordered by lexicographic extension of the subterm relation. □

Proposition 7.5.
If $t \in \mathbb{C}$ and $t \succeq u$ then $u \in \mathbb{C}$.

Proof. We either have $t = u$, but then $u = t \in \mathbb{C}$, or $t \succ u$ in which case $u \in \mathbb{C}$ by the computability property (C_2). □

We conclude this section with a result that is not present in [19], namely a proof of the fact that $\succ$ is decidable.

Theorem 7.6.
Given terms t and u the problem whether $t \succ u$ is decidable.

Proof. Induction on the pair (t, u) ordered by $(\sqsubset, \sqsubset)_{lex}$ followed by a case analysis on t.

- $t = x$. Variables are in normal forms with respect to $\succ$ so we cannot have $x \succ u$.
- $t = @(t_l, t_r)$. Only (H_5) is applicable if $u = @(u_l, u_r)$ and for that, taking typing consideration into account, it is required that $t_l \succeq u_l$, $t_r \succeq u_r$ and $t_l \succ u_l \vee u_l \succ u_r$ all of which are decidable by the induction hypothesis.
- $t = \lambda x : \alpha . t_b$. Only (H_6) is applicable for $u = \lambda x : \alpha . u_b$ and it is required that $t_b \succ u_b$ which we can decide by induction hypothesis.
- $t = f(t_1, \ldots, t_n)$. We have several cases to consider corresponding to application of different clauses of HORPO:
  - (H_1): for every $i \in \{1, \ldots, n\}$ we check whether $t_i \succeq u$ by the application of the induction hypothesis.
  - (H_2): u needs to be of the shape $u = g(u_1, \ldots, u_k)$ with $f \rhd g$ (we assume precedence to be decidable). We need to check whether $t \succ\!\succ \{u_1, \ldots, u_k\}$. So for every $i \in \{1, \ldots, k\}$ we check whether $t \succ u_i$ or $t_j \succ u_i$ for some $j \in \{1, \ldots, n\}$. Typing consideration are helpful in immediate discarding of many cases.
  - (H_3): comparison between all arguments of t and u is decidable by the induction hypothesis so to conclude whether multisets of arguments can be compared we use Theorem 3.3.
  - (H_4): we consider all possible partial flattenings $@(u_1, \ldots, u_k)$ of u (bounded by the size of u) and for each of them we check whether $t \succ\!\succ \{u_1, \ldots, u_k\}$ in the same way as in the (H_2) case.

□

Figure 7.1: Conformance of $\to_\beta$ and $\succ$ to properties required by computability (summarized in Figure 6.1)

Property	**Proof for $\to_\beta$**	**Proof for $\succ$**
(P_1)	Proposition 4.14	Direct from the definition.
(P_2)	Direct from the definition	Direct from the definition.
(P_3)	Proposition 4.15	Proposition 7.3
(P_4)	Direct from the definition	Direct from the definition.
(P_5)	Proposition 4.16	Proposition 7.4
(P_6)	Proposition 4.17	Proposition 7.1
(P_7)	Proposition 4.19	Proposition 7.2
(P_8)	Direct from the definition	Direct from the definition
(P_9)	Direct from the definition	Direct from the definition.

7.3 Well-foundedness of HORPO

In this section we present the proof of well-foundedness of $\succ \cup \to_\beta$. This relation will play an important role in this section so let us abbreviate it by $\rightsquigarrow \equiv \succ \cup \to_\beta$. For the proof we will use the computability predicate proof method due to Tait and Girard which was discussed in Chapter 6.

Note that we will use computability with respect to $\rightsquigarrow$ and for that we need to prove properties (P_1)-(P_9) for $\rightsquigarrow$. In Figure 7.1 conformance of $\to_\beta$ and $\succ$ to those properties is summarized. Note that all those properties easily generalize to the union if they hold for the components.

The crucial lemma states that if function arguments are computable then so is the function application. First we need an auxiliary lemma.

Lemma 7.7.
For any $t = f(t_1, \dots, t_n)$ and $u = g(u_1, \dots, u_k)$ if

(1) $t \succ\!\succ \{u_1, \dots u_k\}$ and

(2) $\forall i \in \{1, \dots, n\}$. $t_i \in \mathbb{C}$ and

(3) $\forall j \in \{1, \dots, k\}$. $t \rightsquigarrow u_j \implies u_j \in \mathbb{C}$,

then $\forall j \in \{1, \dots, k\}$. $u_j \in \mathbb{C}$.

Proof. For a given u_j according to the definition of $\succ\!\succ$ we have two cases:

- $t \succ u_j$, then $u_j \in \mathbb{C}$ by assumption (3).
- $t_i \succeq u_j$ for some i. If $t_i = u_j$ then $t_j \in \mathbb{C}$ by assumption (2) and so $t_i \in \mathbb{C}$. Otherwise $t_i \succ u_j$ but $t_i \in \mathbb{C}$ by (2) and then $u_j \in \mathbb{C}$ by (C_2).

□

Now we can present the aforementioned lemma.

Lemma 7.8.
If $t_1, \ldots, t_n \in \mathbb{C}$ then $t = f(t_1, \ldots, t_n) \in \mathbb{C}$.

Proof. The proof proceeds by well-founded induction on the pair of a function symbol and a multiset of computable terms, $(f, \{\{t_1, \ldots, t_n\}\})$, ordered lexicographically by $(\triangleright, \rightsquigarrow_{mul})_{lex}$. Note that all terms in the multiset are computable and hence strongly normalizable, by (C_1). So $(\triangleright, \rightsquigarrow_{mul})_{lex}$ is well-founded by Theorem 3.15 and Theorem 2.1 which justifies the induction argument.

Since t is neutral we apply (C_3) and we are left to show that $u \in \mathbb{C}$ for an arbitrary u, such that $t \rightsquigarrow u$. We will show that by inner induction on the structure of u. We continue by case analysis on $t \rightsquigarrow u$. The first case corresponds to a beta-reduction step and the following ones to applications of clauses (H_1), (H_2), (H_3) and (H_4) of the HORPO definition. Note that clauses (H_5) and (H_6) are not applicable.

(β) Let $t \rightarrow_\beta u$. By Lemma 4.18 we know that the reduction is in one of the arguments, so for some j we have $u = f(t_1, \ldots t'_j, \ldots t_n)$ with $t_j \rightarrow_\beta t'_j$. For every i, $t_i \in \mathbb{C}$ by assumption and $t'_j \in \mathbb{C}$ by (C_2) so we conclude $u \in \mathbb{C}$ by the outer induction hypothesis.

(H_1) $t_i \succeq u$ for some $i \in \{1, \ldots, n\}$. By assumption $t_i \in \mathbb{C}$ and we either have $t_i = u$ or $t_i \succ u$ but then $u \in \mathbb{C}$ by (C_2).

(H_2) $u = g(u_1, \ldots u_k)$ with $f \triangleright g$. All $u_i \in \mathbb{C}$ for $i \in \{1, \ldots, k\}$ by Lemma 7.7 and we conclude that $u \in \mathbb{C}$ by the outer induction hypothesis since $(f, \{\{t_1, \ldots t_n\}\})$ $(\triangleright, \rightsquigarrow_{mul})_{lex}$ $(g, \{\{u_1, \ldots, u_k\}\})$

(H_3) $u = f(u_1, \ldots u_k)$ with $\{\{t_1, \ldots, t_n\}\} \succ_{MUL} \{\{u_1, \ldots, u_k\}\}$. We can conclude $u \in \mathbb{C}$ by the outer induction hypothesis if we can prove that $u_i \in \mathbb{C}$ for $i \in \{1, \ldots, k\}$. For arbitrary i, by Proposition 3.4 (ii) we get $t_j \succeq u_i$ for some j and since $t_j \in \mathbb{C}$ by assumption we conclude $u_i \in \mathbb{C}$ by (C_2).

(H_4) $@(u_1, \ldots, u_k)$ is some left-partial flattening of u and $t \succ\!\succ \{u_1, \ldots, u_k\}$. By Lemma 7.7 we get $u_i \in \mathbb{C}$ for $i \in \{1, \ldots, k\}$ and hence $u \in \mathbb{C}$.

□

The next step is to show that application of a computable substitution gives computable term, where we define computable substitution as a substitution containing in its domain only computable terms. More formally:

Definition 7.2 (Computable substitution).
We say that $\gamma = [x_1/u_1, \ldots, x_n/u_n]$ is a computable substitution *if for every $i \in \{1, \ldots, n\}$, $u_i \in \mathbb{C}$.*

Lemma 7.9.
Let γ be a computable substitution. Then for any term t, $t\gamma \in \mathbb{C}$.

Proof. We proceed by induction on the structure of t. We have the following cases to consider.

- $t = x$. If $x \in Dom(\gamma)$ then $\gamma = [\ldots, x/u, \ldots]$ and $t\gamma = u$ but $u \in \mathbb{C}$ since γ is a computable substitution. Otherwise $x \notin Dom(\gamma)$ and $t\gamma = x \in \mathbb{C}$ by (C_4).
- $t = f(t_1, \ldots, t_n)$ so $t\gamma = f(t_1\gamma, \ldots, t_n\gamma)$. We apply Lemma 7.8 and we are left to show that for $i \in \{1, \ldots, n\}$, $t_i \in \mathbb{C}$ which easily follows from the induction hypothesis.
- $t = @(t_l, t_r)$ and $t\gamma = @(t_l\gamma, t_r\gamma)$. Both $t_l\gamma$ and $t_r\gamma$ are computable by the induction hypothesis so $t\gamma \in \mathbb{C}$ by the definition of computability.
- $t = \lambda x : \alpha . t_b$ so $t\gamma = \lambda x : \alpha . t_b\gamma$. By application of (C_5) we are left to show that $t_b\gamma[x/u] \in \mathbb{C}$ for any $u \in \mathbb{C}_\alpha$. But $t_b\gamma[x/u] = t_b(\gamma \cup [x/u])$ since $x \notin Dom(\gamma)$. Since $\gamma \cup [x/u]$ is a computable substitution as so is γ and $u \in \mathbb{C}$, we can conclude $t\gamma \in \mathbb{C}$ by the induction hypothesis.

□

Now we are ready to present the main theorem stating that the union of HORPO relation and β-reduction of simply typed λ-calculus, is a well-founded relation on terms.

Theorem 7.10.
The relation $\rightsquigarrow$ is well-founded.

Proof. We need to show that $t \in \mathcal{A}cc$ for an arbitrary t. Consider the empty substitution ϵ, which is computable by definition. We also have $t = t\epsilon$ so we conclude $t \in \mathbb{C}$ by Lemma 7.9 and then $t \in \mathcal{A}cc$ by (C_1). □

Chapter 8

Conclusions

We presented the description of Coq formalization of well-foundedness of the higher-order variant of the recursive path ordering. The development is rather big with 24,190 lines of Coq and 741,543 total characters. It is fully constructive and axiom free. The latter means that all the dependant results had to be formalized as well. Hence the development includes the formalization of multisets, multiset order, simply typed lambda calculus and computability predicate proof method used in the well-foundedness proof of HORPO.

Few comments on the experience with Coq along the course of this development may be in place here. The expressive logic of Coq turned out to be very helpful in this work. In particular dependent types were used extensively and even the very crucial definition of terms was taking advantage of them (see Section 4.1). Main difficulties arisen when the intended equality did not coincide with Coq's Leibniz' equality. The `setoid` tactic makes dealing with such structures somehow easier but still using it requires to prove that every function is compatible with the equivalence relation in question, which does not come for free. Another source of inelegance in the proofs is the lack of any support for handling symmetries, which at times requires many repetitions of (almost) the same argument. Such support, although obviously difficult to realize, would be of great help.

There are various directions in which this work can be extended:

- **Formalization of higher-order rewriting.**
  Our formalization focused on the HORPO ordering and its well-foundedness. Extending this towards general formalization of the higher-order rewriting, shortly introduced in Section 5.2, would be of much interest.

- **Adaptation of the proof to other rewriting frameworks.**
  The HORPO ordering used in this paper is presented for the higher-order rewriting framework of algebraic functional systems (AFSs, see Section 5.2) by Jouannaud and Okada [18]. Another popular format is that of higher order rewriting systems (HRSs) introduced by Nipkow [27]. Van Raamsdonk presented the version of HORPO for HRSs [30]. Formalization of this variant with an attempt to share as much as possible between those two variants would be an interesting goal.

- **Extending the proof for stronger variants of HORPO.**
  Our formalization deals with the simplest definition of HORPO from [19]. In the same publication and later on in [20] and [21] it has been extended and improved in many different ways. Considering (some of) those improvements and extending the definitions and proofs for them is another possible way of continuing this work.

Acknowledgments

The author would like to express his upmost gratitude to the following persons: Femke van Raamsdonk, for suggestion of this research topic as well as her invaluable help in the initial part of this undertaking as the supervisor of my Master Thesis; Hans Zantema, my daily supervisor, for his patience with the fact that the deadline for the finalization of this work has been constantly postponed; Roland Zumkeller for his help with proving uniqueness of derivations in the simply typed λ-calculus in Coq; my thanks also go to the Coq team for keeping up the good work and constantly improving this nice and powerful theorem prover; last but not least my big thanks go to my friends, particularly Jóna Bernadett Eva and Karol Osłowski, who all along believed that this work will one day be finished and were supporting me in the times of despair.

Bibliography

[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] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, 1998.

[6] H. P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science (vol. II)*, pages 117–309, 1992.

[7] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Phd thesis, Université Paris 7, November 1999.

[8] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82:25–49, 2006.

[9] C. Berghofer. A constructive proof of Higman's lemma in Isabelle. In S. Berardi, M. Coppo, and F. Damiani, editors, *Proceedings of the International Workshop Types for Proofs and Programs (TYPES 2003)*, volume 3085 of *LNCS*, pages 66–82. Springer, 2004.

[10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[11] 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.

[12] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[13] S. Coupet-Grimal and W. Delobel. A Constructive Axiomatization of the Recursive Path Ordering. Research report 28-2006, LIF, Marseille, France, January 2006. http://www.lif-sud.univ-mrs.fr/Rapports/28-2006.html.

[14] N. Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[15] The Coq development team. The Coq proof assistant reference manual, version 8.0. `http://pauillac.inria.fr/coq/doc-eng.html`, April 2004.

[16] Brian E. Aydemir et. al. Mechanized metatheory for the masses: The POPLmark challenge. In *TPHOLs*, pages 50–65, 2005.

[17] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 1989.

[18] J.-P. Jouannaud and M. Okada. Executable higher order algebraic specification languages. In *LICS '91*, pages 350–361, 1991.

[19] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *Proceedings of the 14th annual IEEE Symposium on Logic in Computer Science (LICS '99)*, pages 402–411, Trento, Italy, July 1999.

[20] J.-P. Jouannaud and A. Rubio. Higher-order recursive path orderings 'à la carte'. `http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/biblio.html`, 2001.

[21] 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`.

[22] N. de Kleijn. Well-foundedness of RPO in Coq. Master's thesis, Vrije Universiteit, Amsterdam, The Netherlands, August 2003.

[23] A. Koprowski. Well-foundedness of the higher-order recursive path ordering in Coq. Technical Report TI-IR-004, Vrije Universiteit, Amsterdam, The Netherlands, Aug 2004. Master's Thesis.

[24] A. Koprowski. Certified higher-order recursive path ordering. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications, (RTA '06)*, LNCS, Aug 2006. To appear.

[25] F. Leclerc. Termination proof of term rewriting systems with the multiset path ordering: A complete development in the system Coq. 902:312–327, April 1995.

[26] C. Murthy. Extracting constructive content from classical proofs. 1990.

[27] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.

[28] T. Nipkow. An inductive proof of the wellfoundedness of the multiset order. `http://www4.informatik.tu-muenchen.de/~nipkow/misc/index.html`, October 1998. A proof due to W. Buchholz.

[29] H. Persson. Type theory and the integrated logic of programs. May 1999. PhD Thesis.

[30] F. van Raamsdonk. On termination of higher-order rewriting. In A. Middeldorp, editor, *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA '01)*, pages 261–275, Utrecht, The Netherlands, May 2001.

[31] F. van Raamsdonk. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in TCS*, chapter 11, pages 588–668. Cambridge University Press, 2003.

[32] J.-C. Raoult. Proving open properties by induction. *Information Processing Letters*, 29:19–23, 1988.

[33] K. Stoevring, O. Danvy, and M. Biernacka. Program extraction from proofs of weak head normalization. In *Proceedings of the 21st Conference on the Mathematical Foundations of Programming Semantics, (MFPS '05)*, ENTCS, 2005.

[34] W. W. Tait. Intentional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.