Vrije Universiteit, Amsterdam
Faculty of Sciences

Warsaw University
Faculty of Mathematics, Informatics and Mechanics

Master of Science thesis by

Adam Koprowski

Well-foundedness of the higher-order recursive path ordering in Coq

Supervisor:

dr Femke van Raamsdonk
Faculty of Sciences
Vrije Universiteit, Amsterdam

dr Jacek Chrząszcz
Instytut Informatyki
Uniwersytet Warszawski

August 2004

Abstract

This paper presents the proof of termination of the higher-order recursive path order. Both definition of the order and the proof are due to Jouannaud and Rubio. It also describes the development of formalization of that proof in proof checker Coq.

Keywords

term rewriting, termination

Classification

F. Theory of Computation
F.4 Mathematical Logic and Formal Languages
F.4.2 Grammars and Other Rewriting Systems

F. Theory of Computation
F.4 Mathematical Logic and Formal Languages
F.4.1 Mathematical Logic: Lambda calculus and related systems,
Mechanical theorem proving — Coq
Computability theory

Contents

Acknowledgments

This paper is the Master of Science thesis in Computer Science. It was written in Amsterdam during author's stay at the special one year master's program in Computer Science. The program is an agreement between the *Vrije Universiteit* (*Free University Amsterdam*) and author's home university, *Uniwersytet Warszawski* (*Warsaw University*).

In the first place I would like to thank my supervisor, Femke van Raamsdonk. Not only the subject of this thesis was of her authorship but without her constant help and encouragement this work would never come into being. I would also like to thank my supervisor from Poland, Jacek Chrząszcz for his remarks. Last but not least, I would like to thank Roland Zumkeller. At some point in this work I had to prove in Coq uniqueness of derivations on terms of simply typed λ-calculus. After some unsuccessful tries at solving this problem myself I posted question concerning it to Coq club (Coq's mailing list). He proposed solution that is included in this work and for which I'm grateful to him.

Finally I would like to thank all my friends from the dormitory. They did not contribute to the content of this work directly but we have spent together a wonderful time and they made this one year in Amsterdam so special for me that I will never forget it.

Chapter 1

Introduction

Remark. A try has been made to separate the theory from the description of issues strictly related to the Coq development. The latter ones are marked with a gray bar on the margin. Thus reader not interested in details of the development can skip them easily.

1.1. Background

A term rewriting system is said to be terminating if all its reduction sequences are finite. Termination is the most commonly investigated property of term rewriting systems. Although in general it is undecidable many techniques have been developed to prove termination in some particular cases.

One of them uses the recursive path ordering (RPO) defined by Dershowitz. It provides a way to recursively extend a well-founded order on function symbols to order on terms. It is also a reduction order which means that proving $l > r$ for every rewrite rule $l \to r$ is sufficient to conclude termination of given term rewriting system.

Although RPO is well-known for a long time its generalization to higher-order case was missing. The problem was mainly due to the fact that original proof of well-foundedness of the RPO relies on Kruskal's tree theorem and the suitable extension of that theorem to higher-order terms is not known. The breakthrough was introduction of the higher-order recursive path ordering (HORPO) by Jouannaud and Rubio in [6]. Instead of Kruskal's tree theorem it uses the computability predicate proof method due to Tait and Girard.

1.2. Related work

Persson [12] 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 [6]. The proof in [12] is extracted from the classical proof using a minimal bad sequence argument by using open induction due to Raoult [14]. Persson also presents an abstract formalization of well-foundedness of recursive path relations in the proof-checker Agda. The main difference between the work by Persson and the current work is the level of abstraction: here we are much more concrete. Another difference is of course the use of Agda instead of Coq.

Leclerc [9] presents a formalization in Coq of well-foundedness of RPO with the multiset ordering. The Coq script consists of about 250 pages and hence is not presented in full detail in the reference. The focus is on giving upper bounds for descending sequences in RPO.

There are quite some differences between the work by Leclerc and the current work. Most data-types are represented differently, and also the current development is like the one by Persson substantially shorter.

Murthy [10] 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 [3] presents a constructive proof of Higman's lemma in Isabelle. The constructive proof is due to Coquand and Fridlender.

1.3. Overview

We begin by giving a summary of topics discussed in subsequent chapters of this document.

In chapter 2 some basic notions are introduced. In section 2.1 some results concerning relations are mentioned; then in section 2.2 well-foundedness and well-founded induction are discussed, followed by a description of the lexicographic order on a product in section 2.3. Finally, section 2.4 gives a brief description of basic notions in Coq that were needed for the formalization but were missing in the standard library.

Chapter 3 introduces finite multisets and the multiset extension of a relation. Section 3.1 introduces the notion of finite multisets. Section 3.2 describes basic operations on multisets while section 3.3 gives an overview of multisets properties. In section 3.4 the definition of the multiset extension of a relation is given which is then proved to be a strict order in section 3.5 (provided the relation is an order) and to be well-founded in section 3.6 (provided the relation is well-founded).

Chapter 4 focuses on the simply typed λ-calculus. Terms are introduced in section 4.1, followed by description of some basic operations and notions concerning them in section 4.2. Section 4.3 describes formally the definition of substitution on simply typed λ-terms, and in section 4.4 β-reduction is defined.

Chapter 5 gives a very brief description of the computability predicate proof method due to Tait and Girard that is used in the proof of well-foundedness of the higher-order recursive path ordering. The definition of computability is given in section 5.1 and some of its properties are discussed in section 5.2.

Finally, chapter 6 presents the main result: a proof of well-foundedness of the higher-order recursive path ordering. First the higher-order recursive path ordering is defined in section 6.1, and then it is proved to be well-founded in section 6.2.

Of course it is not about numbers but we present some figures to give some feeling of the size of the formalization. The Coq sources consist of 20 files with more than 7000 lines of code and total size around 200 Kb.

The structure of Coq development is presented in figure 1.1. Boxes represent different files with their names, but usually there is a one-to-one correspondence between files and Coq modules and the name of the module is the same as of the file.

There are four main parts:

- *Auxiliaries* with some very basic results being natural extension of the standard library.
- *Multisets* with a development of finite multisets and the multiset extension of a relation.
- *Terms* with the theory of simply typed λ-calculus.
- *Horpo* with definition of the higher-order recursive path ordering and a proof of its well-foundedness.

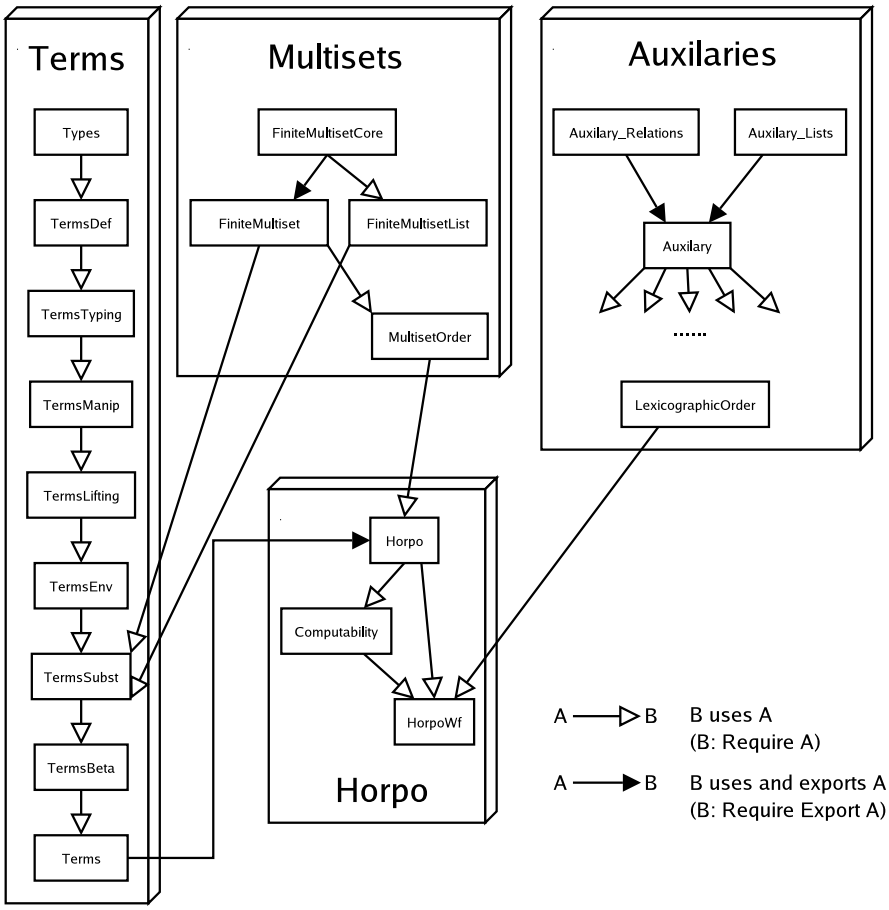

Figure 1.1: Structure of the Coq development

A more detailed description of the content of all the files with references to places in this document where they are discussed can be found in table 1.1

File name	Description	Discussed in
Auxiliaries		
Auxiliary_Relations.v	Few definitions and results concerning relations and modules allowing to deal with equality.	2.4.1
Auxiliary_Lists.v	Some functions and facts concerning manipulation of lists.	2.4.2
Auxiliary.v	Joins two above files adding just few definitions.	
LexicographicOrder.v	Formalization of lexicographic order on a product.	2.3
Multisets		
FiniteMultisetCore.v	Specification of finite multisets.	3.2
FiniteMultiset.v	Finite multiset theory developed using specification.	3.2, 3.3
FiniteMultisetList.v	Implementation of multisets using lists.	3.2
MultisetOrder.v	Multiset extension of a relation.	3.4, 3.5, 3.6
Terms		
Types.v	Notions of simple type and signature	4.1.1, 4.1.2
TermsDef.v	Definitions of environment, preterm, type derivation and typed term	4.1.3, 4.1.4, 4.1.5, 4.1.6
TermsTyping.v	Some results concerning type derivations and typed terms	4.1.6
TermsManip.v	Basic operations and notions concerning terms	4.2
TermsLifting.v	Lifting of terms (corresponding to renaming of variables)	4.3.1
TermsEnv.v	Manipulations on terms' environments	4.3.2
TermsSubst.v	Definition and results concerning substitution	4.3.3
TermsBeta.v	Definition and results concerning Beta-reduction	4.4
Horpo		
Horpo.v	Definition of the higher-order recursive path ordering	6.1
Computability.v	Computability predicate proof method.	5 (5.1, 5.2)
HorpoWf.v	Proof of the well-foundedness of the higher-order recursive path ordering	6.2

Table 1.1: Content of the Coq source files

Chapter 2

Preliminaries

2.1. Relations

We introduce some well-known definitions concerning relations.

Definition 2.1 (Relations)
Let A denote a set and R denote a relation over A (that is: $R \subseteq A \times A$). We will sometimes write $R\ x\ y$ for $(x, y) \in R$. We say that relation A is:

- Reflexive, *if* $\forall x.\ R\ x\ x$.
- Irreflexive, *if* $\forall x.\ \neg R\ x\ x$.
- Transitive, *if* $\forall x, y, z.\ R\ x\ y \wedge R\ y\ z \implies R\ x\ z$.
- Symmetric, *if* $\forall x, y.\ R\ x\ y \implies R\ y\ x$.
- Antisymmetric, *if* $\forall x, y.\ R\ x\ y \wedge R\ y\ x \implies x = y$.
- a strict order, *if it is reflexive and transitive.*
- Well-founded, *if there is no infinite sequence* $R\ a_0\ a_1, R\ a_1\ a_2, \ldots$.

Definition 2.2 (Relations operations)
Let A denote a set and R a relation over A. The P-closure *of R is the least relation extending R that has property P. We list some well-known operations on relations:*

- Reflexive closure *where P states that the relation is reflexive.*
- Transitive closure *where P states that the relation is transitive.*
- Reflexive-transitive closure *where P states that the relation is both reflexive and transitive.*
- Transposition *is the relation defined as* $R^T = \{(x, y) \mid (y, x) \in R\}$.

If a relation is denoted by $\rightarrow$, then by $\rightarrow^*$ we understand its reflexive-transitive closure and by $\rightarrow^+$ its transitive closure.

Most of the definitions presented in this section can be found in Coq library in module Coq.Relations.*. In Coq.Relations.Relation_Definitions most of the results from Definition 2.1 are present, except for irreflexibility and well-foundedness[1]. All results from Definition 2.2 can be found in Coq.Relations.Relation_Operators. Also the proof stating that the transitive closure of a relation is well-founded if the relation itself is, from module Coq.Wellfounded.Transitive_Closure, is being used. Some required results concerning relations were missing though and had to be implemented — they are discussed in section 2.4.

2.2. Well-foundedness, well-founded induction

The notion of well-foundedness proposed in the previous section shows nicely its equivalence with termination, that is absence of infinite descending chains. But as it uses notion of infinite sequences it is not suitable for formalization. Here we give another, more suitable, definition of well-foundedness.

Definition 2.3 (Well-foundedness)
Let A denote a set and $<$ denote a relation over A. The well-founded *part of A, denoted as $\mathcal{W}_A^<$, is inductively defines as follows:*

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

It easily follows that $<$ is well-founded *if $\mathcal{W}_A^< = A$.*

Two induction principles easily follow from this definition.

Definition 2.4 (Well-founded induction principles)
Well-founded induction:

$$\frac{\forall x.\ (\forall y < x.\ P(y)) \implies P(x)}{\forall x.\ P(x)}$$

Well-founded part induction:

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

The notion of well-foundedness corresponding to Definition 2.3 is present in the Coq library in the module Coq.Init.Wf and it was used in our development. The element belonging to set $\mathcal{W}_A^<$ is called *accessible* and definition of accessibility is as follows:

```
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.
```

Then relation is well-founded if all its elements are accessible:

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

Finally both well-founded induction and well-founded part induction are present as:

[1] Which is treated separately in section 2.2

```
well_founded_ind :
  forall (A : Set) (R : A -> A -> Prop),
  well_founded R ->
  forall P : A -> Prop,
  (forall x : A, (forall y : A, R y x -> P y) -> P x) -> forall a : A, P a

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
```

The correspondence with Definitions 2.3 and 2.4 is straightforward.

2.3. Lexicographic order

First we give the definition of lexicographic order on the product of two sets.

Definition 2.5 (Lexicographic order)
Let A and B be two sets ordered by strict orders: $>_A$ and $>_B$ respectively. Then lexicographic order $>_{lex}$ on pairs $A \times B$ is defined as:

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

This definition easily extends to lexicographic order of n sets. It is well-known that the lexicographic product of well-founded strict orders is a well-founded strict order again.

The lexicographic order is present in the Coq library in the module Coq.Relations.Relation_Operators. But it is defined using a dependent product which makes it more difficult to use in our case when we do not need the dependency. That is why in the module LexicographicOrder.v/LexicographicOrder an alternative definition of the lexicographic order of two sets has been given along with proofs that it is a strict order and is well-founded if the two underlying orders are so. Then in the module LexicographicOrder.v/LexicographicOrderTriple a lexicographic order of three sets has been formalized. The composition of two lexicographic orders for two sets has been used exploiting the fact that $A \times B \times C = (A \times B) \times C$. The results for two-fold lexicographic order could be used then.

2.4. Coq preliminaries

There are two Coq files containing some very basic results: Auxiliary_Relations with some definitions and results about relations that are missing in the library and Auxiliary_Lists concerning manipulation on lists. They will be discussed in turn.

2.4.1. Relations

In the file Auxiliary_Relations a few missing notions and results concerning relations are specified.

But the main purpose of modules presented there is to find a way to deal with equality. In Coq equality is defined as Leibniz equality. The problem is that sometimes this does not denote the intended equality. [2] The Setoid mechanism present in Coq is intended to help to deal with such cases. We try to go one step further and to abstract from the type of equality used.

[2] That will be the case for example in implementation of multisets as lists where two non convertible terms can represent the same multiset.

A first step towards this goal is to introduce a module representing a set equipped with any type of equality.

```
Module Type Eqset.
  Parameter A : Set.
  Parameter eqA : A -> A -> Prop.
  Notation "X =A= Y" := (eqA X Y) (at level 70): sets_scope.
  Axiom sid_theoryA : Setoid_Theory A eqA.
End Eqset.
```

There are some additional elements in this module, such as registration of hints to use properties of equality and definitions of scopes for notation. Setoid_Theory is a record stating that relation eqA representing equality is reflexive, symmetric and transitive — it is defined in the standard library in the module Coq.Setoids.Setoid.

Then module Eqset_def is given as a functor taking any set as an argument and building module of type Eqset with standard Coq's Leibniz equality.

```
Module Type SetA.
  Parameter A : Set.
End SetA.
Module Eqset_def (A: SetA) <: Eqset.
  // ...
End Eqset_def.
```

Finally some relation is introduced — it is thought of as an $>$ relation because in most places where it is used that will be the case, but it is not required to be an order so in principle it can be an arbitrary relation.

```
Module Type Ord.
  Parameter A : Set.
  Declare Module S : Eqset with Definition A := A.
  Parameter gtA : A -> A -> Prop.
  Notation "X > Y" := (gtA X Y) : sets_scope.
  Axiom gtA_eqA_compat: forall (x x' y y': A),
    x =A= x' -> y =A= y' -> x > y -> x' > y'.
End Ord.
```

Again, some minor details were omitted in the above listing. Then some additional facts and definitions are build on top of Ord in module OrdFacts.

```
Module OrdFacts (P: Ord).
```

First the relations: $<$, $\leq$, $\geq$ are defined by means of $>$ and $=$. Then $<$ and $>$ are proven to be morphisms with respect to $=$.

As a last step the module type Poset is introduced which extends Ord and requires $>$ to be a strict order.

2.4.2. Lists

The file Auxiliary_Lists contains some useful facts concerning manipulation on lists — a data-type that is widely used through the development. It consists of three sections: ListsGeneral with some general facts concerning library functions manipulating on lists; ListsNth that provide a number of facts about library function nth_error which returns n^{th} element of the list or signals an error if the index is beyond the scope of the list.

```
Definition Exc := option.
nth_error: forall A: Set, list A -> nat -> Exc A.
```

Last section ListsExtras defines three additional functions manipulating on lists and provides some basic results concerning their behavior.

- `Fixpoint initSeg (l: list A) (size: nat) {struct size} : list A := ...`
  returns the first *size* elements of the list (or less if the list is not long enough).
- `Fixpoint seg (l: list A) (from size: nat) {struct from} : list A := ...`
  returns *size* (possibly less if the list is not long enough) elements of the list starting from the element at index *from*.
- `Fixpoint copy (n: nat) (el: A) {struct n} : list A := ...`
  returns list containing n copies of *el*.

Chapter 3

Finite multisets and the extension of a relation to multisets

Multisets will be used in section 6.1 ...

In section 3.1 a brief description of what multisets are and their definition are given, followed by presentation of basic operations on multisets in section 3.2. Then in section 3.3 some properties of multisets are discussed. In section 3.4 two equivalent definitions of an extension of a relation to multisets can be found. The multiset extension being a strict order is proven in section 3.5. Finally well-foundedness of the extension of an relation to finite multisets is shown in section 3.6 — results presented there follow closely the presentation by Nipkow of the proof due to Buchholz [11].

3.1. Finite multisets

Intuitively *"multisets are sets with repeated elements"*. Although this statement is by far too informal for a definition, it gives a good intuition about multisets. More formally:

Definition 3.1 (Multisets)
A multiset M *over a 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* $\mathbb{M}_A$.

We easily recognize that $M(x)$ corresponds to the number of copies of x in M in our intuitive approach. Although most of the theory for multisets holds as well for infinite multisets, the well-foundedness does not and since it is crucial in our development we will restrict our attention to finite multisets only.[1]. So in the rest of the document by multiset we understand finite multiset.

We will use the set-like notation for multisets only using double curly brackets instead of single ones to distinguish from sets. So both by $\{\{a, b, a\}\}$ and by $\{\{a, a, b\}\}$ we understand the function:

$$M(x) = \begin{cases} 2 & \text{if } x = a \\ 1 & \text{if } x = b \\ 0 & \text{otherwise} \end{cases}$$

[1] There are some variants of the definition of finite multisets; some allow for example the multiplicity of an element to be infinite. According to our definition a finite multiset has only finitely many elements and their multiplicities are finite as well.

Multisets are already present in the Coq standard library in the module Coq.Sets.Multiset, but they are defined in a way that allows infinite multisets which, in our case, in unacceptable. As it seemed impossible to easily adopt that definition to deal with the restriction to finite multisets, and also only very basic results are present in the library, the decision has been taken not to use that definition but to provide a new one.

We decided to follow the abstract data type paradigm, which means that the implementation of multisets is hidden and access to them is obtained only through their specification. This decision allowed us to obtain a more neat and easier extensible implementation to the price of more difficult development. Furthermore, the two-layered approach has been used — first there is a module with the specification of basic operations on multisets (to which every implementation has to conform) and then, on top of it, some facts about multisets has been proven (that allows to easily give a new implementation for which all those facts will be valid).

More details about the multisets development can be found in section 3.2 where operations on multisets are discussed.

3.2. Basic operations on multisets

We assume a set A the elements of which are denoted by $x, y, \ldots$. The basic operations on multisets over A with their definitions can be found in table 3.1. The basic primitive operation is multiplicity of x in given multiset M which is simply the value of the function M for x. Then all the other operations are defined by means of multiplicity.

Operation	Notation	Definition
Multiplicity	$M(x)$	
Equality	$M = N$	$M = N \iff \forall x.\ M(x) = N(x)$
Member	$x \in M$	$x \in M \iff M(x) > 0$
Empty multiset	$\emptyset$	$\emptyset(x) := 0$
Singleton	$\{\{x\}\}$	$\{\{x\}\}(x) := 1$ $\{\{x\}\}(y) := 0 \quad x \neq y$
Union	$M \cup N$	$(M \cup N)(x) := M(x) + N(x)$
Intersection	$M \cap N$	$(M \cap N)(x) := min\{M(x), N(x)\}$
Difference	$M \setminus N$	$(M \setminus N)(x) := M(x) \ominus N(x)$ where: $m \ominus n := \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{otherwise} \end{cases}$

Table 3.1: Operations on multisets

As already mentioned in section 3.1, multisets are developed in an abstract data type manner.

First the specification of multisets is given as a module type FiniteMultisetCore.v/FiniteMultisetCore.

```
Module Type FiniteMultisetCore.
  Declare Module Sid: Eqset.
  Parameter eqA_dec: forall (x y: A), {x =A= y}+{~x =A= y}.
   // ...
```

It specifies the multisets over an arbitrary set equipped with an equality relation (Sid of type Eqset) which is assumed to be decidable (eqA_dec). It consists of the operations listed in table 3.2 in the part called "Basic operations", along with their specification equivalent to the formulas in the "Definition" column of table 3.1. It is worth noting that the restriction to finite multisets is due to the fact that, according to the specification, the only way to construct a multiset is by taking an empty multiset (empty) or singleton (singleton).

Operation	Coq definition	Coq notation
Basic operations		
$M(x)$	`mult: A -> Multiset -> nat`	`x/M`
$M = N$	`meq: Multiset -> Multiset -> Prop`	`M =mul= N`
$\emptyset$	`empty: Multiset`	
$\{\{x\}\}$	`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`
Derived operations		
$\{\{x, y\}\}$	`pair: A -> A -> Multiset`	`{{ x, y }}`
$x \in M$	`member: A -> Multiset -> Prop`	`x in M`
$M \cup \{\{x\}\}$	`insert: A -> Multiset -> Multiset`	
$M \setminus \{\{x\}\}$	`remove: A -> Multiset -> Multiset`	

Table 3.2: Operations on multisets in Coq

One additional thing that is needed to work with multisets is an induction scheme. It is of the following shape:

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

So for every property on multisets we can conclude that it holds for all multisets if it holds for empty multiset, and from the fact that it holds for M it is possible to conclude that it holds for union of M and any singleton. This is this axiom that restricts multisets to finite ones.

In the module FiniteMultiset.v/FiniteMultiset we have:

```
Module FiniteMultiset (MC: FiniteMultisetCore).
```

That is it is a functor taking as an argument an implementation of multisets conforming to the specification discussed above, that is, a module of the type FiniteMultisetCore.v/FiniteMultisetCore. It introduces some additional operations on multisets — they are presented in the "Derived operations" part of table 3.2. One not mentioned there[2] is a conversion from list to multiset:

```
Fixpoint list2multiset (l: list A) : Multiset := /* ... */
```

Then using Coq's Setoid mechanism all operations on multisets are proven to be morphisms.[3]. Also a lot of facts about multisets are proven in this module and, because only the specification is used, they are independent from the implementation. They will be discussed in more details in section 3.3.

Finally in the module FiniteMultisetList.v/FiniteMultisetList:

```
Module FiniteMultisetList (ES: Eqset)
 : FiniteMultisetCore with Module Sid := ES.
```

an implementation of multisets represented as lists is given. Two lists are considered to represent the same multiset if one is a permutation of the other. The implementation of the required operations

[2]Because it deals with transformation of data-types in Coq and has no theoretical counterpart.

[3]Basically it means that in all multisets operations one can replace a multiset argument by an equal one (note that we are talking about multisets equality here which clearly differs from Coq's Leibniz' equality). For more information on Coq's Setoids see [4].

on multisets is quite natural (for example the union of multisets corresponds to the concatenation of two lists, and so on). Of course also the proofs that this implementation conforms to the specification need to be given, but, although not always trivial, they are not very interesting from the point of view of this presentation.

3.3. Properties of multisets

Some facts about multisets can be found in table 3.3. They are not intended to form a complete multiset theory in any sense; rather their choice was driven by needs arising from using multisets for our needs. Most of them are trivial[4] so we will not go into details.[5] We will make an exception for the proof of the double_split lemma which is probably the most difficult one and will serve as an example.

Fact 3.1
For all $L, U, D, R \in \mathbb{M}_A$ *the following holds:*

$$L \cup R = U \cup D \implies R = (R \cap D) \cup (U \setminus L)$$

Proof
It might be helpful to imagine that the pairs L, R and U, D are different partitions of some multiset as depicted in figure 3.1. Then using the notation from the picture we have $R \cap D = DR$ and $U \setminus L = UR$ and finally indeed $DR \cup UR = R$.

However one has to be very careful with such an graphical interpretation (known as Venn's diagrams) because although it works well for sets it not necessarily does for multisets. If it was we would have also:

$$L \cup R = U \cup D \implies R = (U \setminus L) \cup (D \setminus L)$$

which obviously does not hold as can be seen by taking $U = D = L = R = \{\{a\}\}$. After this remark we can proceed with the proof itself.

By definition we have:

$$R = (R \cap D) \cup (U \setminus L) \iff$$
$$\forall_x.\ R(x) = min\{R(x), D(x)\} + (U(x) \ominus L(x))$$

Also from the assumption we get:

$$(\bigstar) \quad \forall_x.\ L(x) + R(x) = U(x) + D(x)$$

We proceed by case analysis:

Cases:		**To prove:**
		$R(x) = min\{R(x), D(x)\} + (U(x) \ominus L(x))$
$U(x) \geq L(x)$		$R(x) = min\{R(x), D(x)\} + (U(x) - L(x))$
	$R(x) \geq D(x)$	$R(x) = D(x) + U(x) - L(x)$
		$R(x) = L(x) + R(x) - L(x)$
		trivial
	$R(x) < D(x)$	contradiction with $\bigstar$
$U(x) < L(x)$		$R(x) = min\{R(x), D(x)\}$
	$R(x) \geq D(x)$	contradiction with $\bigstar$
	$R(x) < D(x)$	$R(x) = R(x)$

[4] Some are so trivial that normally they would never be mentioned by any author but such simple facts are also necessary in strict formalization and we included them here for the sake of completeness.

[5] For interested reader the lecture of the Coq file FiniteMultitset.v should be sufficient to understand the proofs.

Lemma name	Lemma content
meq_refl	$M = M$
meq_trans	$\left.\begin{array}{r} M = N \\ N = P \end{array}\right\} \implies M = P$
meq_sym	$M = N \implies N = M$
union_comm	$M \cup N = N \cup M$
intersection_comm	$M \cap N = N \cap M$
union_assoc	$M \cup (N \cup P) = (M \cup N) \cup P$
member_singleton	$x \in \{\{y\}\} \implies x = y$
union_perm	$M \cup N \cup P = M \cup P \cup N$
union_empty	$M \cup \emptyset = M$
not_empty	$\left.\begin{array}{r} M(x) > 0 \\ M = \emptyset \end{array}\right\} \implies \bot$
mult_insert	$(M \cup \{\{a\}\})(a) > 0$
singleton_member	$a \in \{\{a\}\}$
member_member_union	$a \in M \implies a \in (M \cup N)$
member_diff_member	$a \in (M \setminus N) \implies a \in M$
member_union	$a \in (M \cup N) \implies a \in M \vee a \in N$
member_meq_union	$\left.\begin{array}{r} M \cup N = M' \cup N' \\ a \in M \vee a \in N \\ a \notin N' \end{array}\right\} \implies a \in M'$
meq_union_meq	$M \cup P = N \cup P \implies M = N$
meq_meq_union	$M = N \implies M \cup P = N \cup P$
meq_ins_ins_eq	$\left.\begin{array}{r} M \cup \{\{a\}\} = M' \cup \{\{a'\}\} \\ a = a' \end{array}\right\} \implies M = M'$
meq_ins_ins	$M \cup \{\{a\}\} = M' \cup \{\{a'\}\} \implies M = (M' \cup \{\{a'\}\}) \setminus \{\{a\}\}$
member_ins_ins_meq	$\left.\begin{array}{r} M \cup \{\{a\}\} = M' \cup \{\{a'\}\} \\ a \neq a' \end{array}\right\} \implies a \in M'$
meq_ins_rem	$a \in M \implies M = (M \setminus \{\{a\}\}) \cup \{\{a\}\}$
meq_diff_meq	$M = N \implies M \setminus P = N \setminus P$
diff_MM_empty	$M \setminus M = \emptyset$
ins_meq_union	$\left.\begin{array}{r} M \cup \{\{a\}\} = N \cup P \\ a \in N \end{array}\right\} \implies M = (N \setminus \{\{a\}\}) \cup P$
mem_memrem	$\left.\begin{array}{r} a \neq b \\ a \in M \end{array}\right\} \implies a \in (M \setminus \{\{b\}\})$
member_notempty	$x \in M \implies M \neq \emptyset$
singleton_notempty	$\{\{x\}\} \neq \emptyset$
union_isempty	$M \cup N = \emptyset \implies M = \emptyset$
union_notempty	$M \neq \emptyset \implies M \cup N \neq \emptyset$
member_insert	$a \in (M \cup \{\{a\}\})$
double_split	$L \cup R = U \cup D \implies R = (R \cap D) \cup (U \setminus L)$
partition	$\forall_{P,Q \subset A}.\ (\forall_{x:A}.\ P(x) \vee Q(x)) \implies$ $\exists_{M_p, M_q}.\ \left\{\begin{array}{l} M = M_p \cup M_q \\ (\forall_p.\ p \in M_p \implies P(p)) \\ (\forall_q.\ q \in M_q \implies Q(q)) \end{array}\right.$

Table 3.3: Facts about multisets from module FiniteMultiset.v/FiniteMultiset

□

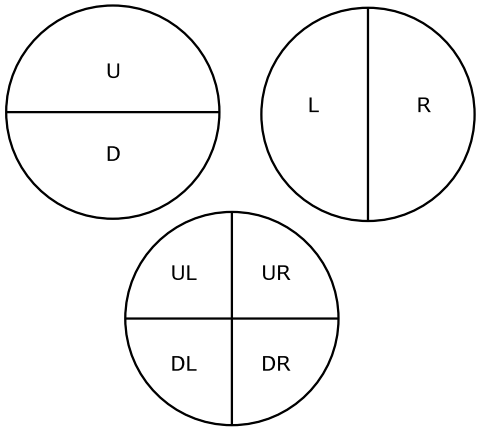

Figure 3.1: Diagram for Fact 3.1

All the facts from table 3.3 are proven in the module FiniteMultiset.v/FiniteMultiset. There are some additional facts dealing with the conversion between multisets and lists, namely:

```
Fact member_list_multiset: forall l,
  forall x, In x l -> member x (list2multiset l).
Fact member_multiset_list: forall l,
  forall x, member x (list2multiset l) ->
  (exists2 y, In y l & x =A= y).
Fact list2multiset_app: forall M1 M2,
  list2multiset (M1 ++ M2) =mul= (list2multiset M1) + (list2multiset M2).
```

Most of the facts are easy so they will not be discussed here.

3.4. Definition of the extension of a relation to multisets

We start with a definition of multiset reduction.

Definition 3.2 (Multiset reduction, $\rhd_{mul}$)
Let $>$ be a relation on A. The corresponding multiset reduction *relation ($\rhd_{mul}$) on $\mathbb{M}_A$ is defined as:*

$$M \rhd_{mul} N \text{ iff } \begin{cases} \text{there exist } X, Y \in \mathbb{M}_A \text{ and } a \in A \text{ such that:} \\ \quad M = X \cup \{\{a\}\} \\ \quad N = X \cup Y \\ \quad \forall_{y \in Y}.\ a > y \end{cases}$$

We will say then that the triple $(X, a, Y)_{mul}$ proves $M \rhd_{mul} N$. $(X, a, Y)_{mul}$ will also be called a witness of the fact that $M \rhd_{mul} N$.

The definition of the extension of a relation to multisets follows.

Definition 3.3 (Multiset extension of a relation, $>_{mul}$)
The multiset extension of a relation *($>_{mul}$) is the transitive closure of the multiset reduction relation ($\rhd_{mul}$).*

Remark. Often we assume the underlying order $>$ on the set A to be a strict order and that is why multiset extension of a relation will be sometimes referred to as a multiset order or as a multiset extension of an order. But our definition deals as well with the case when the relation $>$ on A is not necessarily an order — that will be the case in the definition of the higher-order recursive path ordering in section 6.1.

This is just one way of defining the multiset extension. Another one is to combine the number of steps leading to the ordering in one explicit definition:

Definition 3.4 (Multiset extension of a relation — alternative definition, $>_{MUL}$) *Let $>$ be a relation on A. We define the multiset extension of a relation ($>_{MUL}$) as:*

$$M >_{MUL} N \text{ iff } \begin{cases} \text{exist } X, Y, Z \in \mathbb{M}_A \text{ such that:} \\ \quad Y \neq \emptyset \\ \quad M = X \cup Y \\ \quad N = X \cup Z \\ \quad \forall_{z \in Z}.\ \exists_{y \in Y}.\ y > z \end{cases}$$

We will say then that the triple $(X, Y, Z)_{MUL}$ proves $M >_{MUL} N$.

Later on we will show that those two definitions are equivalent, in case the relation $>$ on A is an order, but we need to collect some facts about them first.

A first easy fact states that there is no multiset less than an empty one.

Fact 3.2

$$\forall_{M \in \mathbb{M}_A}.\ \neg(\emptyset >_{mul} M)$$

Proof
First notice is that a similar result for the multiset reduction $\rhd_{mul}$ obviously holds — for the reduction relation to hold the proving element from $\emptyset$ is needed but there are no elements at all in $\emptyset$.

For $\emptyset >_{mul} M$ to hold either $\emptyset \rhd_{mul} M$ or $\emptyset \rhd_{mul} M' >_{mul} M$ would have to be true while both are not according to the former notice.

□

A next step is to prove transitivity of the relation $>_{MUL}$ from Definition 3.4.

Lemma 3.3 (Transitivity of $>_{MUL}$)

$$\forall_{M,N,P}.\ \left.\begin{array}{l} M >_{MUL} N \\ N >_{MUL} P \end{array}\right\} \implies M >_{MUL} P$$

Proof
From the definition of $>_{MUL}$ for $M >_{MUL} N$ and $N >_{MUL} P$ we have following:

$$\left.\begin{array}{llll} (1) & M = X_1 \cup Y_1 & (1') & N = X_2 \cup Y_2 \\ (2) & N = X_1 \cup Z_1 & (2') & P = X_2 \cup Z_2 \\ (3) & Y_1 \neq \emptyset & (3') & Y_2 \neq \emptyset \\ (4) & \forall_{y \in Z_1}.\ \exists_{x \in Y_1}.\ x > y & (4') & \forall_{y \in Z_2}.\ \exists_{x \in Y_2}.\ x > y \end{array}\right\} (\bigstar)$$

We claim that the triple $(X_1 \cap X_2,\ Y_1 \cup (Y_2 \setminus Z_1),\ Z_2 \cup (Z_1 \setminus Y_2))$ proves $M >_{MUL} P$. We proceed by showing in turn that all conditions from Definition 3.4 are satisfied.

- $Y_1 \cup (Y_2 \setminus Z_1) \neq \emptyset$.
  Trivial since $Y_1 \neq \emptyset$.

- $M = (X_1 \cap X_2) \cup (Y_1 \cup (Y_2 \setminus Z_1))$.
  We show that this holds by a sequence of equivalent formulas. The comment on the right explains how the step has been obtained.[6] It is good to remark that the most difficult part of the proof is accomplished by application of Fact 3.1 (double_split) proven in section 3.1.

$$\begin{array}{rcr} M = (X_1 \cap X_2) \cup (Y_1 \cup (Y_2 \setminus Z_1)) & \iff & /\text{by (1)}/ \\ X_1 \cup Y_1 = (X_1 \cap X_2) \cup (Y_1 \cup (Y_2 \setminus Z_1)) & \iff & /\text{by union\_assoc \& union\_perm}/ \\ X_1 \cup Y_1 = (X_1 \cap X_2) \cup (Y_2 \setminus Z_1) \cup Y_1 & \iff & /\text{by meq\_meq\_union}/ \\ X_1 = (X_1 \cap X_2) \cup (Y_2 \setminus Z_1) & \iff & /\text{by double\_split}/ \\ Z_1 \cup X_1 = Y_2 \cup X_2 & \iff & /\text{by union\_comm}/ \\ X_1 \cup Z_1 = X_2 \cup Y_2 & \iff & /\text{by (2) and (1')}/ \\ N = N & & /\textit{trivial}/ \end{array}$$

- $N = (X_1 \cap X_2) \cup (Z_2 \cup (Z_1 \setminus Y_2))$.
  Analogously.

- $\forall_{y \in Z_2 \cup (Z_1 \setminus Y_2)}.\ \exists_{x \in Y_1 \cup (Y_2 \setminus Z_1)}.\ x > y$
  We proceed by case analysis depending on whether $y \in Z_2$ or $y \in (Z_1 \setminus Y_2)$.

  $y \in Z_2$
  : By (4') we have $x \in Y_2$ with $x > y$. Then we have two cases. Either $x \in Z_1$ and then by (4) there exists $x' \in Y$ with $x' > x$ which is the witness for the proof: $x' \in Y_1 \cup (Y_2 \setminus Z_1)$ because $x' \in Y_1$ and $x' > y$ because $x' > x > y$. The second case is that $x \notin Z_1$, but then we can use x because $x \in Y_2$ and $x \notin Z_1$ implies $x \in (Y_2 \setminus Z_1)$ and $x > y$.

  $y \in (Z_1 \setminus Y_2)$
  : Then $y \in Z_1$ and by (4) we conclude that there is $x \in Y_1$ with $x > y$ which trivially satisfy two main conclusions of the lemma.

□

Remark. In the above proof of transitivity of $>_{MUL}$ we used the fact that relation $>$ on set A is transitive. As we are going to use this lemma for the following proof of equivalence of Definitions 3.3 and 3.4 it will not cover the case of multiset extension of an arbitrary relation. But indeed those definitions differ for the multiset extension of relation that is not transitive as we can see by taking $A = \{a, b, c\}, > = \{(a, b), (b, c)\}$. Then $a >_{mul} c$ because $a \rhd_{mul} b$ and $b \rhd_{mul} c$. But $a >_{MUL} c$ does not hold as $a > c$ does not hold.

Now we are ready to prove the equivalence of Definitions 3.3 and 3.4.

Theorem 3.4

$$\forall_{M,N \in \mathbb{M}_A}.\ M >_{mul} N \iff M >_{MUL} N$$

[6] The identifiers correspond to formulas from table 3.3 and numbers to assumptions from (★).

Proof

($\Rightarrow$)

Induction on the derivation of $M >_{mul} N$. If $M >_{mul} N$ holds then either $M \rhd_{mul} N$ or $M >_{mul} M' >_{mul} N$. If $M \rhd_{mul} N$ then we have the witness $(X, a, Y)_{mul}$ but then obviously $(X, \{\{a\}\}, Y)_{MUL}$ proves $M >_{MUL} N$. If in turn $M >_{mul} X >_{mul} N$ then we apply the induction hypothesis to conclude $M >_{MUL} M' >_{MUL} N$ and then finally by Lemma 3.3 (stating that $>_{MUL}$ is transitive) we have $M >_{MUL} N$.

($\Leftarrow$)

By the definition of $>_{MUL}$ the following formulas are satisfied:

$$\begin{array}{llll} (1) & M = X \cup Y & (3) & Y \neq \emptyset \\ (2) & N = X \cup Z & (4) & \forall_{z \in Z}.\ \exists_{y \in Y}.\ y > z \end{array}$$

We proceed by induction on Y. The base case with $Y = \emptyset$ leads to a contradiction since then by (3) we would have $\emptyset \neq \emptyset$.

The induction hypothesis states that:

$$(IH) : \forall_{M,N,X,Z \in \mathbb{M}_A} \left\{ \begin{array}{l} Y \neq \emptyset \\ M = X \cup Y \\ N = X \cup Z \\ \forall_{z \in Z}.\ \exists_{y \in Y}.\ y > z \end{array} \right. \implies M >_{mul} N$$

and what is to be proven is:

$$\forall_{M,N,X,Z \in \mathbb{M}_A} \left\{ \begin{array}{ll} Y \cup \{\{a\}\} \neq \emptyset & (1) \\ M = X \cup (Y \cup \{\{a\}\}) & (2) \\ N = X \cup Z & (3) \\ \forall_{z \in Z}.\ \exists_{y \in (Y \cup \{\{a\}\})}.\ y > z & (4) \end{array} \right. \implies M >_{mul} N$$

We distinguish two cases.

- $Y = \emptyset$
  Then triple $(X, a, Z)_{mul}$ trivially proves $M >_{mul} N$.
- $Y \neq \emptyset$
  This case is more involved. By applying fact partition from table 3.3 we split Z into two depending on the relation of the elements with a, that is:

$$\exists_{Z_>, Z_< \in \mathbb{M}_A}.\ \left\{ \begin{array}{l} Z = Z_> \cup Z_< \\ \forall_{z \in Z_>}.\ a > z \\ \forall_{z \in Z_<}.\ a <= z \end{array} \right.$$

Then we claim that $M >_{mul} X \cup (Z_< \cup \{\{a\}\}) >_{mul} N$ which by transitivity of $>_{mul}$ completes the proof. We show that both reductions hold.

- $M >_{mul} X \cup (Z_< \cup \{\{a\}\})$
  This can be shown by applying an instance of the induction hypothesis with $X := X \cup \{\{a\}\}$, $Z := Z_<$, $M := M$, $N := X \cup (Z_< \cup \{\{a\}\})$. All the premises but the last one are trivially satisfied. For $\forall_{z \in Z_<}.\ \exists_{y \in Y}.\ y > z$ we use assumption (4) (we

can since $Z_< \subset Z$). Then we have some $y > z$, but it comes from $Y \cup \{\{a\}\}$ and we need it to be from Y. But we can easily show that case when $y = a$ leads to contradiction since then $y = a > z$ by assumption (4) and $y = a <= z$ by definition of $Z_<$.

- $X \cup (Z_< \cup \{\{a\}\}) >_{mul} N$
  Easily proven by the triple $(X \cup Z_<, a, Z_>)_{mul}$.

□

There is one more useful fact about the multiset extension of a relation.

Fact 3.5
If $M >_{mul} N$ *then* $\forall n \in N.\ \exists m \in M.\ m >^+ n \vee m = n$

Proof
Fact that if $M \rhd_{mul} N$ then $\forall n \in N.\ \exists m \in M.\ m \geq n$ follows easily from the definition of multiset reduction. Then the result to be proven is an easy consequence, as $>_{mul}$ is a transitive closure of $\rhd_{mul}$.

□

Note that obviously if $>$ is transitive then the statement $m >^+ n \vee m = n$ in the above fact can be simply replaced by $m \geq n$.

The multiset reduction and multiset extension of a relation are formalized in module MultisetOrder.v/MultisetOrder:

```
Module MultisetOrder (MC: FiniteMultisetCore).
```

A first idea was to give an underlying order $>$ on the set as another parameter to this functor. But this turned out to be problematic. First because we also want to use multiset extension of arbitrary relation, not only order, so it would be impossible to instantiate this module in that case. But even more serious problem is due to the fact that in section 6.1, where the higher-order recursive path ordering is defined, its multiset extension is used in one of the clauses defining the relation. That means that multiset extension has to be combined in one mutually recursive definition with definition of extended relation. This is unattainable in current version of Coq. That is why a more traditional approach was used where abstraction over some parameters is done through the sections mechanism instead of modules. So we have:

```
Variable gtA : A -> A -> Prop.
Let leA x y := ~gtA x y.

Notation "X >A Y" := (gtA X Y) (at level 50).
Notation "X <=A Y" := (leA X Y) (at level 50).

Variable gtA_dec: forall (a b: A), {a >A b}+{a <=A b}.
Variable gtA_so: strict_order gtA.
Variable gtA_eqA_compat: forall (x x' y y': A),
 x =A= x' -> y =A= y' -> x >A y -> x' >A y'.
```

Although the notation of `gtA` suggests that this is an order and it is assumed to be so by `gtA_so`, but Coq's section mechanism takes care that the assumption `gtA_so` will be required only in those lemmas where it is really employed — for the remaining ones `gtA` can be an arbitrary relation.

Then the multiset reduction ($\rhd_{mul}$, Definition 3.2) is defined as:

```
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.
```

and the definition of multiset extension ($>_{mul}$, Definition 3.3) follows.

```
Definition MultisetGt := clos_trans Multiset MultisetRedGt.
```

Also alternative definition of multiset extension ($>_{MUL}$, Definition 3.4) is formalized as:

```
Inductive MultisetGT (M N: Multiset) : Prop :=
| MSetGT: forall X Y Z,
   X <>mul empty ->
   M =mul= Z + X ->
   N =mul= Z + Y ->
   (forall y, y in Y -> (exists2 x, x in X & x >A y)) ->
   MultisetGT M N.
```

It is easy to recognize the definitions presented in this section in those Coq counterparts.

3.5. Multiset extension of an order being strict order

By now we have collected quite a lot facts about multisets and the multiset order. We need just one more, and then the proof of the fact that $>_{mul}$ is a strict order is straightforward.

Remark. In this section we restrict attention to the multiset extension of an order.

Fact 3.6

$$M >_{mul} N \iff M \cup P >_{mul} N \cup P$$

Proof
If we can show that:

$$M >_{mul} N \iff M \cup \{\{a\}\} >_{mul} N \cup \{\{a\}\} \tag{3.1}$$

then we can prove the main goal easily proceeding by induction on P using this fact.

Now we proceed with the proof of (3.1).

($\Rightarrow$)
According to Theorem 3.4 it is possible to use $>_{mul}$ and $>_{MUL}$ interchangeably. From the assumption that $M >_{mul} N$ it follows that $M >_{MUL} N$ for which we have a witness: $(X, Y, Z)_{MUL}$. It is an easy observation that $(X \cup \{\{a\}\}, Y, Z)_{MUL}$ proves $M \cup \{\{a\}\} >_{mul} N \cup \{\{a\}\}$.

($\Leftarrow$)
This time we have a witness $(X, Y, Z)_{MUL}$ for $M \cup \{\{a\}\} >_{MUL} N \cup \{\{a\}\}$. If $a \in X$ then obviously $(X \setminus \{\{a\}\}, Y, Z)_{MUL}$ proves $M >_{MUL} N$ which ends the proof. If $a \notin X$ then clearly $a \in Y$ and $a \in Z$ and then $(X, Y \setminus \{\{a\}\}, Z \setminus \{\{a\}\})_{MUL}$ proves $M >_{MUL} N$.

□

The main result of this section follows.

Theorem 3.7
If $>$ is a strict order then so is $>_{mul}$.

Proof

Ir-reflexivity:
Using Facts 3.6 and 3.2 we have:

$$M >_{mul} M \iff M \cup \emptyset >_{mul} M \cup \emptyset \iff \emptyset >_{mul} \emptyset \implies \bot$$

Transitivity:
Follows immediately from the definition of $>_{mul}$. Another way of reasoning is that $>_{mul}$ is equivalent to $>_{MUL}$ (Theorem 3.4) which was proven to be transitive (Lemma 3.3).

□

3.6. Well-foundedness of the multiset extension of a well-founded relation

Remark. In this section only employed property of relation $>$ on A is its well-foundedness, which means that the results hold as well for the multiset extension of any well-founded relation.

We will begin by proving a simple fact that will be used later.

Fact 3.8

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

Proof
We distinguish two cases depending on the fact if the element from $M \cup \{\{a\}\}$ proving that it is bigger from N is a or not.

- If it is, then $N = M \cup Y$ where $\forall_{y \in Y}.\ y < a$. So the right disjunct of the hypothesis is trivially satisfied by Y.
- If it is not, then $N = X \cup Y$, $M \cup \{\{a\}\} = X \cup \{\{a_0\}\}$ and $\forall_{y \in Y}.\ y < a_0$. Then $M' = Y \cup (M \setminus \{\{a_0\}\})$ satisfies the left disjunct. The first condition holds because $X = (M \setminus \{\{a_0\}\}) \cup \{\{a\}\}$ (note that $a \neq a_0$). For the second condition we have to show that $Y \cup (M \setminus \{\{a_0\}\}) \lhd_{mul} M$ which can be easily proven using the triple $(M \setminus \{\{a_0\}\}, a_0, Y)_{mul}$.

□

Lemma 3.9

$$\left.\begin{array}{rr} (1) & \forall_{b<a}\ \forall_{M \in \mathcal{W}^{\lhd}_{\mathbb{M}}}\ M \cup \{\{b\}\} \in \mathcal{W}^{\lhd}_{\mathbb{M}} \\ (2) & M_0 \in \mathcal{W}^{\lhd}_{\mathbb{M}} \\ (3) & \forall_{M \lhd_{mul} M_0}\ M \cup \{\{a\}\} \in \mathcal{W}^{\lhd}_{\mathbb{M}} \end{array}\right\} \implies M_0 \cup \{\{a\}\} \in \mathcal{W}^{\lhd}_{\mathbb{M}}$$

Proof
By the definition of $\mathcal{W}^{\lhd}_{\mathbb{M}}$ we have to show that:

$$\forall_N.\ N \lhd_{mul} M_0 \cup \{\{a\}\} \implies N \in \mathcal{W}^{\lhd}_{\mathbb{M}}$$

By Fact 3.8 we have following two cases why $N \lhd_{mul} M_0 \cup \{\{a\}\}$ holds:

- $N = M \cup \{\{a\}\}$ for some $M \lhd_{mul} M_0$. Then $N \in \mathcal{W}^{\lhd}_{\mathbb{M}}$ by (3).
- $N = M_0 \cup K$ and $\forall_{k \in K}.\ k < a$. Then we proceed by induction on K.

  Induction base $K = \emptyset$, then $N = M_0$ and we get $N \in \mathcal{W}^{\lhd}_{\mathbb{M}}$ by (2).

  Induction step $K = K_0 \cup \{\{k\}\}$. We have $k < a$ and by induction hypothesis $K_0 \in \mathcal{W}^{\lhd}_{\mathbb{M}}$, so $N \in \mathcal{W}^{\lhd}_{\mathbb{M}}$ follows immediately from (1).

□

Lemma 3.10

$$(\forall_{b<a}\ \forall_{M \in \mathcal{W}^{\lhd}_{\mathbb{M}}}\ M \cup \{\{b\}\} \in \mathcal{W}^{\lhd}_{\mathbb{M}}) \implies (\forall_{M \in \mathcal{W}^{\lhd}_{\mathbb{M}}}\ M \cup \{\{a\}\} \in \mathcal{W}^{\lhd}_{\mathbb{M}})$$

Proof
Application of well-founded part induction followed by the use of Lemma 3.9 — first two premises are among assumptions of this lemma and the third one is induction hypothesis.

□

Lemma 3.11

$$\forall_{a \in \mathcal{W}^{<}_{A}}.\ \forall_{M \in \mathcal{W}^{\lhd}_{\mathbb{M}}}.\ M \cup \{\{a\}\} \in \mathcal{W}^{\lhd}_{\mathbb{M}}$$

Proof
Well-founded part induction on a using Lemma 3.10.

□

It is common to assume well-foundedness of underlying order $<$ on A and from that to conclude that $\lhd_{mul}$ on $\mathbb{M}_A$ is well-founded as well. Here a slightly different approach is used. In the following theorem it is shown that M belongs to well-founded part of $\mathbb{M}_A$ with $\lhd_{mul}$ if all its elements belong to well-founded part of A with $<$. Then formerly mentioned result follows easily. This approach allows us to use well-founded part induction on multisets (for instance when $<$ is not well-founded on the whole A or when it is not known in advance).

Lemma 3.12

$$\forall_{a \in M}.\ a \in \mathcal{W}^{<}_{A} \implies M \in \mathcal{W}^{\lhd}_{\mathbb{M}}$$

Proof
Induction on M.

Induction base To prove $\emptyset \in \mathcal{W}^{\lhd}_{\mathbb{M}}$. By definition of $\mathcal{W}^{\lhd}_{\mathbb{M}}$ it is left to show: $\forall_M.\ M \lhd_{mul} \emptyset \implies M \in \mathcal{W}^{\lhd}_{\mathbb{M}}$ which is trivial because there is no multiset less than $\emptyset$.

Induction step To prove $M \cup \{\{a\}\} \in \mathcal{W}_{\mathbb{M}}^{\lhd}$. After applying Lemma 3.11 it is left to show that firstly $a \in \mathcal{W}_A^{<}$, but that follows from assumption that all elements of M are in $\mathcal{W}_A^{<}$, and secondly that $M \in \mathcal{W}_{\mathbb{M}}^{\lhd}$, which in turn is an easy consequence of the induction hypothesis.

□

The following fact expresses the usual situation — relation $<$ on set A is well-founded and then so is the multiset reduction $\lhd_{mul}$ on $\mathbb{M}_A$.

Fact 3.13

$$\forall_{a \in A}.\ a \in \mathcal{W}_A^{<} \implies \forall_{M \in \mathbb{M}_A}.\ M \in \mathcal{W}_{\mathbb{M}}^{\lhd}$$

Proof
Follows directly from Lemma 3.12

□

It is time to present the main result of this chapter. So far all the results concerned with multiset reduction $\lhd_{mul}$, but what we are really interested in is the multiset extension $<_{mul}$. The following theorem is the equivalent of Lemmas 3.12 and 3.13 but this time for the multiset extension of a relation.

Theorem 3.14 (Well-foundedness of multiset extension of a relation)

- $\forall_{a \in M}.\ a \in \mathcal{W}_A^{<} \implies M \in \mathcal{W}_{\mathbb{M}}^{<_{mul}}$
- $\forall_{a \in A}.\ a \in \mathcal{W}_A^{<} \implies \forall_{M \in \mathbb{M}_A}.\ M \in \mathcal{W}_{\mathbb{M}}^{<_{mul}}$

Proof
It is an easy observation that if element was in well-founded part of the set with a given relation then it will remain in well-founded part for transitive closure of that relation. As multiset extension $<_{mul}$ is transitive closure of multiset reduction $\lhd_{mul}$ first fact follows from Lemma 3.12. The second fact in turn is an easy consequence of the first one.

□

Formalization of facts from this chapter can be found in section MultisetOrder_Wf of module MultisetOrder.v/MultisetOrder.

Thanks to the strictly formal nature of Nipkow's publication [11], which is used as the base of presentation in this section, it is very well suited for verification. Moreover the granulation level of the facts is very fine so there is one to one correspondence between the results presented here and the Coq lemmas; they are also very concise (10 lines at most) so for interested reader it should be of no difficulty to follow them.

Chapter 4

Simply typed λ-calculus

The λ-calculus is a language to express functions, and the evaluation of functions applied to arguments. It is well-known and among its many applications it is usually used as a meta-language of higher-order term rewriting systems. Later, in chapter 6, the higher-order recursive path ordering will be introduced as a relation on terms of the simply typed *lambda*-calculus.

In section 4.1 terms of simply typed λ-calculus are defined. Then in section 4.2 some terminology and some operations on those terms are discussed. Section 4.3 is devoted to the subject of substitution concept. Finally in section 4.4 beta-reduction of simply typed λ-calculus is presented.

4.1. Terms of the simply typed λ-calculus

4.1.1. Simple types

We begin with the usual inductive definition of *simple types*.

Definition 4.1 (Simple types)
Assuming a set of base types, *we define* simple types *as follows:*

- *Every base type is a simple type.*
- *If A and B are simple types then $A \to B$ is also a simple type.*

So a type is either a *base type* or is of the shape $A \to B$ with A and B simple types; we call it an *arrow type* then. We assume the usual notational conventions about writing simple types.

Then we introduce a congruence on types as follows:

Definition 4.2 (Congruence on types)
The congruence on types *($\equiv$) is generated by identifying all basic types. We say that $A \equiv B$ if*

- *either A and B are both simple types,*
- *or $A = A' \to A''$, $B = B' \to B''$, $A' \equiv B'$ and $A'' \equiv B''$.*

Note that two types are equivalent if they have the same arrow structure.

Simple types are introduced in the module type Types.v/SimpleTypes. First an arbitrary non-empty set of base types with decidable equality is assumed:

```
Module Type SimpleTypes.
  Parameter BaseType: Set.
  Parameter eq_BaseType_dec: forall (a b: BaseType), {a=b}+{~a=b}.
  Parameter baseTypesNotEmpty: BaseType.
```

Then the usual inductive definition of simple types follows:

```
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.
```

Finally, the definition of equivalence on types can be found in the module TermsDef.v/TermsDef. Although logically it belongs to the module SimpleTypes, in the present version of Coq every definition given in the module type has to be repeated in its implementation so, to avoid this, it was more convenient to move it to the module dealing with terms. The definition is as expected:

```
Fixpoint TypeEq (A B: SimpleType) { struct A } : Prop :=
match A, B with
| A1-->A2, B1-->B2 => TypeEq A1 B1 /\ TypeEq A2 B2
| #_, #_ => True
| _, _ => False
end.
Notation "A == B" := (TypeEq A B) (at level 50).
```

Also functions `isBaseType: SimpleType -> Prop` and `isArrowType: SimpleType -> Prop` stating that given type is a base type or an arrow type respectively can be found in this module.

4.1.2. Signature

Now, we introduce the usual notion of signature.

Definition 4.3 (Signature)
A signature *(Σ) consists of a set of function symbols, equipped with their types, that is of pairs $f : \delta$, where δ is a type.*

In our development we abstract from signature making it a module type which is given as a parameter to the module defining terms. It consists of an arbitrary, non-empty set of function symbols equipped with decidable equality and a function assigning a type to every function symbol.

```
Module Type Signature.
  Declare Module ST : SimpleTypes.
  Parameter FunctionSymbol: Set.
  Parameter eq_FunctionSymbol_dec: forall (f g: FunctionSymbol), {f=g}+{~f=g}.
  Parameter functionSymbolsNotEmpty : FunctionSymbol.
  Parameter f_type: FunctionSymbol -> SimpleType.
End Signature.
```

4.1.3. Preterms

We proceed with the usual definition of preterms.[1].

[1] In literature they are often referred to as pseudo-terms or raw-terms.

Definition 4.4 (Preterms)
The set of preterms *over given signature* Σ *and set of variables* $\mathcal{X}$ *is defined by the following grammar:*

$$\mathcal{T} := x \mid f \mid @(\mathcal{T}, \mathcal{T}) \mid \lambda x : A.\ \mathcal{T}$$

The grammar rules for preterms denote a variable, function symbol, application and abstraction, respectively. Application is assumed to be associative to the left and we will often write $u_1(u_2, \ldots, u_n)$ instead of $(u_1, \ldots, u_n)$ especially if u_1 is a function symbol.

This is the first place where our development differs substantially from [6]. First Jouannaud and Rubio assign the arity to a function symbol which is reflected in its type (which is a product type: $f : \delta_1 \times \ldots \times \delta_n \to \delta$, where n is an arity of f) and in the function symbol rule of preterms which becomes $f(\mathcal{T}, \ldots, \mathcal{T})$. The reason why we do not follow this approach is the will to follow the standard definitions of simply typed λ-calculus. Thanks to that this part of the development makes just a general foundations for development of theory of simply typed λ-calculus and is somehow independent of its application for the higher-order recursive path ordering.

The second difference is the presence of the type of the variable in abstraction, which was missing in [6], but has been added in its extended version [7]. We will explain this decision later, while discussing the typed terms.

Preterms are defined in module TermsDef.v/TermsDef. The most important development decision that had to be made here was use of named variables versus deBruijn indices. The main advantage of the former choice is its better readability, on the other hand latter choice allows to easily deal with α-conversion problems. Let us remind that, roughly speaking, α-conversion rule identifies terms that differ only on variable names. Usually while working with simply typed λ-calculus one does not distinguish α-convertible terms, which corresponds to the idea that choice for variable names does not matter. But in strictly formal development dealing with α-conversion rule is not trivial and gives rise to a lot of technical problems. That is why we have chosen to use deBruijn indices instead. The idea is to give numbers to variables instead of names. Then if we think about a term as a tree, the index of a variable indicates by which abstraction it is bound counting from its occurrence in the direction to the root of the tree; if it is greater than the number of abstractions on the way to the root then it is free. The inductive definition of preterms is then as follows:

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

4.1.4. Environments

For typing purpose we introduce environments.

Definition 4.5 (Environment)
Environment *is a finite set of declarations for distinct variables:*

$$\Gamma = \{x_1 : \delta_1, \ldots, x_n : \delta_n\}$$

where x_i *is a variable and* δ_i *is a type for that variable* ($x_i \neq x_j$ *for* $i \neq j$).

The set of variables declared in environment is denoted as:

$$Var(\Gamma) = \{x_1, \ldots, x_n\}$$

Because of the use of deBruijn indices environment is simply a list of simple types. Then the type of a variable can be found in environment at index equal to the number assigned to the variable.

```
Definition Env := list (option SimpleType).
Definition EmptyEnv : Env := nil.
```

Environment is represented as a list of `option SimpleType` because apart of normal declarations for variables there can be also dummy variables (represented by `None`).

```
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).
```

`VarD E x A` states that in the environment E the variable x has type A and `VarUD E x` that the variable x has no type assigned in E.

```
Definition decl A E := Some A::E.
Definition declDummy E := None::E.
```

Finally `decl A E` returns the environment E extended with declaration of a variable of type A and, similarly, `declDummy E` extends E with a declaration of dummy variable.

4.1.5. Type derivations

We present the typing inference system for the simply typed λ-calculus

Definition 4.6 (Type derivations for the simply typed λ-calculus)
Type system for the simply typed λ-calculus over a signature Σ consists of following rules:

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

$$\frac{\Gamma \vdash s : \delta \to \gamma \qquad \Gamma \vdash t : \delta}{\Gamma \vdash @(s,t) : \gamma} \qquad \frac{\Gamma \cup \{x : \delta\} \vdash s : \gamma}{\Gamma \vdash s : \delta \to \gamma}$$

Here again we differ from the presentation in [6], where there is one more rule, called the congruence rule, which makes it possible to identify all basic types and thus collapse them into single sort. The authors remark there that this allows for a smoother technical development but we do not consider it to be a considerable simplification and we prefer to keep distinction between different sorts in our inference rules.

It is easy to recognize the clauses from Definition 4.6 in the following definition from module TermsDef.v/TermsDef.

```
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,
          (decl A E |- Pt := B) ->
          (E |- \A => Pt := A --> B)
| TApp: forall E A B Pt0 Pt1,
          (E |- Pt0 := A --> B) ->
          (E |- Pt1 := A) ->
          (E |- Pt0 @@ Pt1 := B)
where "E |- Pt := A" := (Typing E Pt A).
```

So `Typing` is a relation taking as arguments an environment, a preterm and a simple type and represents a derivation showing that in that environment the given preterm has the given type.

4.1.6. Typed terms

Now we are ready to present the definition of typed terms.

Definition 4.7 (Typed terms)
A term s has type δ in the environment Γ if $\Gamma \vdash s : \delta$ is provable in the interference system from Definition 4.6.
A term s is typable in the environment Γ if there exists type δ such that s has type δ in the environment Γ.
A term s is typable if it is typable in some environment Γ.
The set of simply typed λ-terms *consists of all typable terms.*

The definition of typed terms can be found in module `TermsDef.v/TermsDef`. It is of the shape:

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

So it consists of an environment, a preterm, a type and a derivation showing that the former three components are correct, in the sense that in that environment, that preterm has the given type.

Some facts about typed terms are proven in module `TermsTyping.v/TermsTyping`. They represent the well-known facts about simply typed λ-calculus . The most important ones are:

- Uniqueness of derivations. That is if two terms have the same environment, preterm and type components then their typing component is also the same and thus they are equal.

  ```
  Theorem deriv_uniq: forall M N,
    env M = env N -> term M = term N -> type M = type N -> M = N.
  ```

- Uniqueness of typing. That is if we have: $E \vdash P : \delta$ and $E \vdash P : \delta'$ then $\delta = \delta'$.

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

- Decidability of equality on terms.

  ```
  Lemma eq_Term_dec : forall (M N: Term), {M=N}+{M<>N}.
  ```

- Decidability of the problem: given an environment E and a preterm P whether exists type δ such that $E \vdash P : \delta$ (decidable in linear time in the sum of the sizes of E and P).

  ```
  Definition autoType E Pt :
    {N: Term | env N = E & term N = Pt} +
    {~exists N: Term, env N = E /\ term N = Pt}.
  Proof.
    (* ... *)
  Defined.
  ```

4.2. Basic operations on terms

We will collect some notions and notations useful for the rest of the presentation in the following definition.

Definition 4.8

- *For application* $@(l, r)$*,* l *is called* left application argument *and* r right application argument.
- *Let us consider a compound application of the shape* $@(u_1, \ldots, u_n)$ *or equivalently* $u_1(u_2, \ldots, u_n)$*, which is obtained by composition of the application, that is:* $@(@(\ldots @(u_1, u_2)\ldots), u_n)$*. Then* u_1 *is referred to as* application head, $u_2, \ldots, u_n$ *as* application arguments *and any of the two, that is* $u_1, \ldots, u_n$ *as* application units
- Function application *is an application which has a function symbol as its head.*
- $@(@(u_1, \ldots, u_i), u_{i+1}, \ldots, u_n)$ *is called a* (partial) left-flattening *of* $@(u_1, \ldots, u_n)$ *if* $i \neq n$*.*
- *Term is* neutral *if it is not an abstraction.*
- *For a term* $\lambda x : A,\ P$*, the subterm* P *is called* abstraction body *and* A abstraction type.

We proceed with definition of sub-term relation.

Definition 4.9 (Sub-term, $\sqsubset$, $\sqsubseteq$)
The inductive definitions of sub-term *(*$\sqsubseteq$*) and* strict sub-term *(*$\sqsubset$*) relations are recursively defined as follows:*

$$\frac{N \sqsubseteq L}{N \sqsubset @(L, R)} \qquad \frac{N \sqsubseteq R}{N \sqsubset @(L, R)} \qquad \frac{N \sqsubseteq P}{N \sqsubset \lambda x : A,\ P}$$

$$\frac{N = M}{N \sqsubseteq M} \qquad \frac{N \sqsubset M}{N \sqsubseteq M}$$

Obviously the strict sub-term relation is well-founded and well-founded induction with its use (called induction on structure of term) will be used in many proofs dealing with terms.

Module TermsManip.v/TermsManip introduces all the results and notions presented in this section plus a lot of simple facts about terms. We will discuss them very briefly here omitting some minor results and concentrating on definitions that are crucial to understand the rest of the presentation.

- `isVar`, `isFunS`, `isAbs`, `isApp` all of type `Term -> Prop` state that a given term is a variable, a function symbol, an abstraction and an application respectively.
- `absBody: forall M, isAbs M -> Term` gives the abstraction body and `absType: forall M, isAbs M -> SimpleType` returns the abstraction type.
- for an application `appBodyL: forall M, isApp M -> Term` returns the left application argument and `appBodyR: forall M, isApp M -> Term` returns the right application argument.
- There are two functions dual to those mentioned in the two previous points: `buildAbs` and `buildApp`. They take as arguments all necessary components and proofs that some required constraints between components hold and build abstraction and application respectively.
- `appUnits: Term -> list Term` return all application units of a given term, `appHead: Term -> Term` returns application head and `appArgs: Term -> list Term` returns application arguments.

  There is no restriction for an argument to be an application. For a term that is not an application we assume that application head is the term itself, list of arguments is empty and application units consist only of the term itself.
- `isArg` and `isAppUnit` both of type `Term -> Term -> Prop` state that the first argument is an application argument or application unit respectively of the second argument.
- `isNeutral: Term -> Prop` state that a term is neutral.

- `isFunApp: Term -> Prop` state that a term is a function application.
- `isPartialFlattening: list Term -> Term -> Prop` takes a list of terms $@(u_1, \ldots, u_n)$ and states that this is a partial flattening of term given as the second argument (note that u_1 might be an application).

Then a sub-term relation is introduced as:

```
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.
```

Then this relation is proved to be well-founded.

```
Lemma subterm_wf: well_founded subterm.
```

Some results about this relation follow like the fact that an application unit of M is a sub-term of M (appUnit_subterm) or that if $@(u_1, \ldots, u_n)$ is a partial flattening of M then u_i is a sub-term of M for any i (partialFlattening_subterm). Well-founded induction with use of sub-term relation is also used in many proofs, like for instance in proof that if all application units of two terms are equal then the terms are equal (eq_units_eq_terms). There is much more than that but as most of the facts are easy from theoretical point of view we are not going into details here.

4.3. Substitution

The purpose of this section is to introduce the notion of substitution. The length of this presentation might be surprising at first but the formalization of substitution is far more complex than one might expect. The reason for that is that we are acquainted with the substitution on level of preterms. But if we want to reason about typed terms this is not sufficient — we have to define substitution on typed terms which means that we have to take environments into account. This turns out not to be easy. For example it is obvious that for the substitution to be correct some prerequisites for substituted terms have to hold; but spelling them out is not that obvious.

The material presented in this section is based on [7]. It is described there in very formal and complete manner so we will skip most of the proofs and refer interested reader to that position.

In section 4.3.1 we discuss the process of lifting terms. Lifting is an operation equivalent to renaming of free variables but for the variant when variables are represented using deBruijn indices. Next in section 4.3.2 we introduce some operations on environments that will be used later on in section 4.3.3 where the actual definition of substitution and some results concerning it are presented.

4.3.1. Lifting of terms

Using deBruijn indices a variable is represented as a number and its type can be found by looking at environment at that index. This number changes depending on the nesting level of abstractions because we use the convention that variable number 0 is bounded by the most nested abstraction.[2]. That is why for the substitution the lifting operation is needed. Lifting is a relocation of variables numbers, sometimes also referred as introducing dummy variables.

Remark. Throughout the formalization we used named variables to increase the readability but as this section is only relevant for representation of variables as deBruijn indices we will use this notation in this section, so the abstraction will be written as $\lambda A.\ M$, without variable name.

Definition 4.10 (Lifting)
By $M\uparrow_k^n$ we denote operation on terms that leaves its first k variables unchanged and increases the numbers of the rest by n. It is inductively defined as follows:

$$\begin{aligned} f\uparrow_k^n &= f \\ x\uparrow_k^n &= \begin{cases} x+n & \text{if } x \geq k \\ x & \text{otherwise} \end{cases} \\ @(L,\ R)\uparrow_k^n &= @(L\uparrow_k^n,\ R\uparrow_k^n) \\ \lambda A.\ M\uparrow_k^n &= \lambda A.\ M\uparrow_{k+1}^n \end{aligned}$$

$M\uparrow^n$ stands for $M\uparrow_0^n$.

In previous definition we described behavior of lifting operation on preterms. Now we will prove that this behaves as expected on typed terms.

Lemma 4.1 (Lifting)
If $\Gamma \vdash M : \delta$ then $\Gamma' \vdash M\uparrow_k^n : \delta$ where Γ' is obtained from Γ by taking first k variables unchanged then adding n dummy variables and then the remaining part of Γ.

Proof
By induction on the structure of term M.

□

First the operation of lifting on preterms is introduced (Definition 4.10):

```
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.
```

[2] or the closest one on the way to the root if we think about terms as trees

As a next step the shape of environment obtained after lifting a term is formalized (`liftedEnv` and the result of lifting a typed term (`lift`; Lemma 4.1).

```
Definition liftedEnv (n: nat) (E: Env) (k: nat) : Env :=
  initSeg E k ++ copy n None ++ seg E k (length E).

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).
```

4.3.2. Operations on environments

We begin with the definition of composition and subtraction operations on environments.

Definition 4.11

- *Given two environments* Γ *and* Λ *their* composition *is the environment* $\Gamma \cdot \Lambda = \Lambda \cup \{x : \delta \in \Gamma \mid x \notin Var(\Lambda)\}$.
- *Given two environments* Γ *and* Λ *the result of* subtraction *operation is the environment* $\Gamma \setminus \Lambda = \{x : \delta \in \Gamma \mid x \notin Var(\Lambda)\}$.

Now we proceed with crucial definition of compatibility between environments.

Definition 4.12 (Compatibility)
We will say that environments Γ *and* Λ *are* compatible for variable x *if*

$$\left.\begin{array}{l} x : \delta \in \Gamma \\ x : \delta' \in \Lambda \end{array}\right\} \Longrightarrow \delta = \delta'$$

which means that if variable is declared in both environments then it is declared with the same type.

Then we say that environments Γ *and* Λ *are* compatible *(denoted as* $\Gamma \bowtie \Lambda$*) if they are compatible for every variable* x*.*

Now we show that terms can be typed in extended environment.

Lemma 4.2

- *If* $\Gamma \vdash M : \delta$ *then* $\Lambda \cdot \Gamma \vdash M : \delta$ *for any environment* Λ.
- *If* $\Gamma \vdash M : \delta$ *and* $\Gamma \bowtie \Lambda$ *then* $\Gamma \cdot \Lambda \vdash M : \delta$.

Proof
By induction on structure of term M.

□

Results from this section are formalized in module `TermsEnv.v/TermsEnv`.

First definition of composition of two environments (`env_compose`) and subtraction (`env_subtract`) are given (Definition 4.11).

```
Fixpoint env_compose (E1 E2: Env) {struct E1} : Env :=
  match E1, E2 with
  | nil, nil => nil
  | L, nil => L
  | nil, L => L
  | _::E1, Some a::E2 => Some a :: env_compose E1 E2
  | e1::E1, None::E2 => e1 :: env_compose E1 E2
  end.
Notation "E1 [+] E2" := (env_compose E1 E2) (at level 50, left associativity).

Fixpoint env_subtract (G: Env) (H: Env) {struct G} : Env :=
match G, H with
| nil, _ => EmptyEnv
| G, nil => G
| None :: G', _ :: H' => None :: env_subtract G' H'
| Some g :: G', None :: H' => Some g :: env_subtract G' H'
| Some g :: G', Some h :: H' => None :: env_subtract G' H'
end.
Notation "E1 [-] E2" := (env_subtract E1 E2) (at level 50, left associativity).
```

Then compatibility of two environments on a given variable is defined as `env_comp_on` and compatibility of two environments as `env_comp` (Definition 4.12).

```
Definition env_comp_on E1 E2 x : Prop :=
  forall A B,
    E1 |= x := A ->
    E2 |= x := B ->
    A = B.

Definition env_comp E1 E2 : Prop :=
  forall x, env_comp_on E1 E2 x.
Notation "E1 [<->] E2" := (env_comp E1 E2) (at level 70).
```

Finally the results of Lemma 4.2 are formalized as `typing_ext_env_l` and `typing_ext_env_r`.

4.3.3. Definition of substitution

We begin with the definition of substitution.

Definition 4.13 (Substitution)
A substitution *is a finite set of pairs: variable name and typed term to be substituted for that variable.*

$$\gamma = \{x_1/\Gamma_1 \vdash M_1 : \delta_1, \ldots, x_n/\Gamma_n \vdash M_n : \delta_n\} \quad (x_i \neq x_j \text{ for } i \neq j)$$

The domain of the substitution *is the environment*

$$Dom(\gamma) = \{x_1 : \delta_1, \ldots, x_n : \delta_n\}$$

The range of the substitution *is the environment*

$$Ran(\gamma) = \bigcup_{i \in \{1,\ldots,n\}} \Gamma_i$$

We denote by $\gamma_{\setminus V}$ *the* restriction of the substitution *obtained by excluding* V *from the domain of* γ.

If we want to define the application of a substitution γ to a term M in a sensible way, some additional conditions have to hold.

Definition 4.14 (Compatibility of substitution)
A substitution $\gamma = \{x_1/\Gamma_1 \vdash M_1 : \delta_1, \ldots, x_n/\Gamma_n \vdash M_n : \delta_n\}$ is said to be compatible with a term $\Lambda \vdash N : \sigma$ if the following conditions are satisfied:

- $\forall_{i,j \in \{1,\ldots,n\}, i \neq j}.\ \Gamma_i \bowtie \Gamma_j$
- $Dom(\gamma) \bowtie \Lambda$
- $Ran(\gamma) \bowtie \Lambda \setminus Dom(\gamma)$

A substitution can be applied to a term only if they are compatible.

Remark. In [7] only the last two conditions of the above definition are included in the definition of compatibility of substitution; the first one is a part of the definition of substitution itself. We moved it here for the sake of readability of the Coq development.

The definition of substitution operation on preterms follows.

Definition 4.15 (Substitution operation)
The result of applying substitution γ to term M, with γ compatible with M is inductively defined as follows:

$$
\begin{aligned}
f\gamma &= f \\
x\gamma &= \begin{cases} x & \text{if } x \notin Var(\gamma) \\ M & \text{if } x/\Gamma \vdash M : \delta \in \gamma \end{cases} \\
@(M,N)\gamma &= @(M\gamma, N\gamma) \\
(\lambda x : A.\ M)\gamma &= \lambda x : A.\ M\gamma_{\setminus x}
\end{aligned}
$$

The following lemma verifies that the above definition makes sense for substitution operating on typed terms.

Lemma 4.3
Assume substitution γ compatible with term $\Gamma \vdash M : \delta$.
Then $\Gamma \cdot Ran(\gamma) \vdash M\gamma : \delta$.

Proof
By induction on a structure of term M.

□

The following lemma shows the correspondence between the environments of terms from a substitution with the range of this substitution.

Lemma 4.4
Assume term M and substitution $\gamma = \{x_1/\Gamma_1 \vdash M_1 : \delta_1, \ldots, x_n/\Gamma_n \vdash M_n : \delta_n\}$ compatible with it. Then the following conditions hold:

- *$Ran(\gamma) = Ran(\gamma) \cdot \Gamma_i$, for any $i \in \{1, \ldots, n\}$.*
- *$Ran(\gamma) \vdash M_i : \delta_i$, for any $i \in \{1, \ldots, n\}$.*

Content of this section corresponds to Coq module TermsSubst.v/TermsSubst.

Substitution is defined as:[3]

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

First a number of simple definitions and notations for checking if a variable is in the substitution domain is given.

```
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).

Definition varIsSubst (G: Subst) x : Set :=
  {T: Term | G |-> x/T}.
Notation "G |-> x /*" := (varIsSubst G x) (at level 50, x at level 0).

Definition varIsNotSubst (G: Subst) x : Prop :=
  nth_error G x = None \/ nth_error G x = Some None.
Notation "G |-> x /-" := (varIsNotSubst G x) (at level 50, x at level 0).
```

So `G |-> x/T` holds if $x/T \in G$, `G |-> x/*` if $x \in Var(Dom(G))$ and `G |-> x/-` if $x \notin Var(Dom(G))$.

Then there are definitions of substitution domain and range (subst_dom and subst_ran respectively).

```
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.
```

`subst_envs_comp` is a formalization of the first condition from Definition 4.14 and `correct_subst` holds if a substitution is compatible with a term.

```
Definition subst_envs_comp (G: Subst) : Prop :=
  forall i j Ti Tj,
    i <> j ->
    G |-> i/Ti ->
    G |-> j/Tj ->
    env Ti [<->] env Tj.
Record correct_subst (M: Term) (G: Subst) : Set := {
  envs_c:
    subst_envs_comp G;
  dom_c:
    subst_dom G [<->] env M;
  ran_c:
    subst_ran G [-] subst_dom G [<->] env M
}.
```

[3] Actually this definition is in the module TermsLifting.v/TermsLifting as some basic results concerning lifting applied to substitution are given there.

Then Definition 4.15 is translated to:

```
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
      | None => Var i
      | Some None => Var i
      | Some (Some Q) => term (lift Q l)
      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.
```

It is worth noting that for performance reasons all the terms in a substitution are not lifted by one after every encountered abstraction but there is an index (l) keeping count of how many abstraction has been passed and only when it is necessary the term from substitution is lifted by that value.

Finally the main result of this section: definition of substitution (Lemma 4.3).

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

Lemma 4.4 is formalized as `subst_comp_env` and `typing_in_subst_env`.

But the most difficult and time-consuming turned out to be a proof of the fact that:

$$M\ \{x_1/T_1\}\ \{x_2/T_2, \ldots, x_n/T_n\} = M\ \{x_1/T_1, x_2/T_2, \ldots, x_n/T_n\}$$

First it had to be proven that if two substitutions on the left-hand side are compatible then their union used on the right-hand side of the equation is compatible with M and this alone was far from trivial (`subst_disjoint_c`). Then the equality of obtained terms was shown (`subst_disjoint`).

4.4. Beta-reduction

The β-reduction relation is one of the crucial definitions in theory of simply typed λ-calculus. It expresses how the application of a function to an argument is evaluated.

Definition 4.16 (β-reduction , $\rightarrow_\beta$)
The β-reduction relation *($\rightarrow_\beta$) is defined as the smallest relation on terms that satisfies:*

$$(\lambda x : A,\ M)N \rightarrow_\beta M[x/N]$$

and is moreover closed under term formation. More explicitly it is inductively defined as follows:

$$\frac{}{(\lambda x : A,\ M)N \to_\beta M[x/N]} \qquad \frac{M \to_\beta M'}{\lambda x : A,\ M \to_\beta \lambda x : A,\ M'}$$

$$\frac{M \to_\beta M'}{@(M, N) \to_\beta @(M', N)} \qquad \frac{M \to_\beta M'}{@(N, M) \to_\beta @(N, M')}$$

The main result concerning β-reduction we are going to need is the behavior of β-reduction applied to function application. We state it without a proof here.

Fact 4.5

$$f(u_1, \ldots, u_i, \ldots, u_n) \to_\beta f(u_1, \ldots, u_i', \ldots, u_n) \quad \textit{where} \quad u_i \to_\beta u_i'$$

Definition and results concerning β-reduction are in the module TermsBeta.v/TermsBeta. For the definition of β-reduction first we need to show that the substitution used there is well-formed.

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

Then the definition of β-reduction relation is as follows:

```
Inductive BetaRed : Term -> Term -> Prop :=
| BetaStep: forall M (Mapp: isApp M) (MLabs: isAbs (appBodyL Mapp)),
               M -b-> subst (beta_subst M Mapp MLabs)
| BetaAbs:  forall M M' (Mabs: isAbs M) (M'abs: isAbs M'),
               absBody Mabs -b-> absBody M'abs ->
               M -b-> M'
| BetaAppL: forall M N (Mapp: isApp M) (Napp: isApp N),
               appBodyL Mapp -b-> appBodyL Napp ->
               appBodyR Mapp = appBodyR Napp ->
               M -b-> N
| BetaAppR: forall M N (Mapp: isApp M) (Napp: isApp N),
               appBodyR Mapp -b-> appBodyR Napp ->
               appBodyL Mapp = appBodyL Napp ->
               M -b-> N
  where "M -b-> N" := (BetaRed M N).
```

The result stated as Fact 4.5 is proved in two variants as:

```
Fact app_beta_args: forall M N f,
  isApp M ->
  M -b-> N ->
  term (appHead M) = ^f ->
  exists Mb, exists Nb,
    isArg Mb M /\
    isArg Nb N /\
    Mb -b-> Nb /\
    insert Mb (list2multiset (appArgs N)) =mul=
    insert Nb (list2multiset (appArgs M)).
```

In such version we do not capture the fact that terms u_i and u_i' from Fact 4.5 (named here as Mb and Nb respectively) are at the same position in function application before and after β-reduction , but we are not going to use that observation. In fact all we will need is the slightly weaker variant, namely:

```
Fact app_beta_args_eq: forall M N Q f,
  isApp M ->
  M -b-> N ->
  term (appHead M) = ^f ->
  isArg Q N ->
  (isArg Q M \/ (exists2 Mb, Mb -b-> Q & isArg Mb M)).
```

Chapter 5

Computability

In this chapter we present the computability predicate proof method due to Tait and Girard. Only a short overview is given with no more than necessary details for the understanding of the presentation in the following chapters. More details can be found in [5]. In section 5.1 we present definition of computability and in section 5.2 some properties of computability are discussed.

5.1. Definition of computability

In this section we present definition of computability predicate.

Definition 5.1 (Computability)
A simply typed λ-term t is computable with respect to relation on terms $\gg$ if one of the following clauses holds:

- *t has a basic type and is strongly normalizable with respect to $\gg$.*
- *t has an arrow type $\delta \to \gamma$ and $@(t, n)$ is computable for every computable term n of type δ.*

Note that it is usual to assume that variables are computable. We do not do that following the presentation of [7]. Computability of variables follows from the Definition 5.1 and it will be stated as one of computability properties.

The part of the development concerning computability can be found in the module Computability.v/Computability. First the definition of computability with respect to arbitrary relation R is introduced there:

```
Variable R : Term -> Term -> Prop.
Notation "X <-R- Y" := (R Y X) (at level 50).
Notation "X -R-> Y" := (R X Y) (at level 50).
Let Rtrans := clos_trans Term R.
Notation "X -R*-> Y" := (Rtrans X Y) (at level 50).

Definition AccR := Acc (transp Term R).
```

```
Fixpoint ComputableS (E: Env) (Pt: Preterm) (T: SimpleType) {struct T} : Prop :=
  exists W: E |- Pt := T,
    match T with
    | #_ =>
       AccR (buildT W)
    | Tl --> Tr =>
       forall Pt',
         ComputableS E Pt' Tl ->
         ComputableS E (Pt @@ Pt') Tr
    end.

Definition Computable M := ComputableS M.(env) M.(term) M.(type).

Definition AllComputable Ts := forall T, In T Ts -> Computable T.
```

5.2. Computability properties

We will now give a list of computability properties used in the proof of well-foundedness of the higher-order recursive path ordering given in section 6.2. All the properties are standard.

Lemma 5.1 (Computability properties)
In the following properties computable is meant to be computable with respect to $\gg$.

(1) Variables are computable.

(2) Every computable term is strongly normalizable (with respect to $\gg$*).*

(3) If t *is computable and* $t \gg s$ *then* s *is computable.*

(4) If t *is neutral then* t *is computable iff* s *is computable for every* $t \gg s$.

(5) If s *and* t *are both computable then* $@(s,t)$ *is computable.*

(6) $\lambda x : A,\ t$ *is computable if for every computable* s, $t[x/s]$ *is computable.*

(7) If s *is computable then* $s{\uparrow^i}$ *is computable for every* i.

(8) If $\Gamma \vdash M : \delta$ *is computable then* $\Lambda \cdot \Gamma \vdash M : \delta$ *is computable for any environment* Λ.

From the above properties we easily conclude the following:

Fact 5.2

(9) If t *is computable and* $t \gg^+ s$ *then* s *is computable.*

(10) If $@(t_1, \ldots, t_n)$ *is a partial flattening of* t *and all* t_i *are computable for* $i \in \{1, \ldots, n\}$ *then* t *is computable.*

(11) If $s = @(s_1, \ldots, s_k)$ *and* s_i *is computable for every* $i \in \{1, \ldots, k\}$ *then* s *is computable.*

Property (10) in theory is trivial because partial flattening is just a different way of looking at the same term. But in the development the representation of partial flattening is different than the term as it is a list of all application arguments, so this last statement has to be proved and in fact the proof is non-trivial.

Properties of computability make up the only part of the development that is left unproved. All properties mentioned in Fact 5.1 are stated as axioms. Then derived properties listed in Fact 5.2 are proven using them.

The table below lists names of the lemmas corresponding to the computability properties.

Property	Name of Coq lemma
(1)	`comp_var`
(2)	`comp_imp_acc`
(3)	`comp_step_comp`
(4)	`neutral_comp_step`
(5)	`comp_app`
(6)	`comp_abs`
(7)	`comp_lift`
(8)	`comp_ext_env`
(9)	`comp_manysteps_comp`
(10)	`comp_pflat`
(11)	`comp_units_comp`

Chapter 6

Higher-order recursive path ordering

In a sense this is the main chapter of this document as it presents the actual definition of the higher-order recursive path ordering and the proof of its well-foundedness — all previous chapters introduced all the necessary apparatus for that purpose. The version of the higher-order recursive path ordering is the simplest one presented by Jouannaud and Rubio in [6]. In the same document and later in [7] it has been extended in various ways.

We begin by first giving the definition of the higher-order recursive path ordering in section 6.1. The proof of this relation being well-founded follows in section 6.2.

6.1. Definition of the higher-order recursive path ordering

Definition of the higher-order recursive path ordering (or HORPO for short) is as follows:

Definition 6.1 (The higher-order recursive path ordering, $\succ$)
We assume a well-founded order on the set of function symbols, called a precedence, denoted by $\triangleright$. We proceed with definition of the higher-order recursive path ordering ($\succ$) which is an order on terms. By $\succeq$ we denote its reflexive closure and by $\succ_{mul}$ its multiset extension.

For two terms $\Gamma \vdash M : \delta$ and $\Sigma \vdash N : \delta'$ we have $M \succ N$ if $\delta \equiv \delta'$ and one of the following holds:

1. $M = f(m_1, \ldots, m_k)$
   there exists $i \in \{1, \ldots, k\}$ *such that* $m_i \succeq N$.
2. $M = f(m_1, \ldots, m_k)$
   $N = g(n_1, \ldots, n_l)$
   $f \triangleright g$
   $M \succ\!\succ \{n_1, \ldots n_l\}$
3. $M = f(m_1, \ldots, m_k)$
   $N = f(n_1, \ldots, n_l)$
   $\{\{m_1, \ldots m_k\}\} \succ_{mul} \{\{n_1, \ldots, n_l\}\}$
4. $@(n_1, \ldots, n_l)$ *is a partial flattening of* N
   $M \succ\!\succ \{n_1, \ldots n_l\}$
5. $M = @(m_l, m_r)$
   $N = @(n_l, n_r)$
   M *is not a functional application*
   $\{\{m_l, m_r\}\} \succ_{mul} \{\{n_l, n_r\}\}$

6. $M = \lambda x : B,\ M'$
   $N = \lambda x : B',\ N'$
   $M' \succ N'$

where $\succ\!\!\succ$ *is a relation between a term and a set of terms, defined as:*
$M \succ\!\!\succ \{N_1, \ldots, N_l\}$ *if for all* $i \in \{1, \ldots, l\}$ *we have:*
if $type(M) \equiv type(N_i)$ *then* $M \succ N_i$
if $type(M) \not\equiv type(N_i)$ *then there exists* M' *such that:*
M' *is an application argument of* M *and* $M' \succeq N_i$

There are few differences with the definition given in [6].

The first one is that we use one clause less. That is because we use only multiset extension of $\succ$ to compare two functional applications with the same head symbol, whereas in the original version also a lexicographic extension is used. The reason for that are simply time constraints of this work but it should be relatively easy to extend it with this additional clause.

The second difference is that we use deterministic version of $\succ\!\!\succ$, while in [6] non-deterministic statement is given with a remark that this can be replaced by a deterministic equivalent.[1]

The third, and last, difference is in the 5th clause. In the original version presented in [6] the condition that M is not a functional application was not present. We added it and we claim, although we will not prove it formally, that this can be done without loosing power of an order as this case was redundant. The idea is that first for typing reasons in clause 5 m_l can only be compared with n_l and m_r with n_r. Thus if we have $f(m) \succ @(n_1, n_2)$ only few cases are possible. First, it can happen that $f = n_l$ but then $m \succ n_2$ and the whole order can be shown using clause 3 instead of 5. The other case is when $f \succ n_l$ and $m \succeq n_r$. It is more involved but then $f \succ n_l$ could be obtained only with the use of clause 2 or 4 and it is also possible to do without resorting to clause 5.

The definition of HORPO can be found in the module `Horpo.v/Horpo`. First precedence is formalized as a module type:

```
Module Type Precedence.
  Declare Module S: Types.Signature.
  Declare Module P: Poset with Definition A := S.FunctionSymbol.
  Parameter Ord_wf: well_founded (transp A gtA).
End Precedence.
```

The above definition reflects the fact that the precedence is a strict order over function symbols from the signature that is in addition well-founded.

Then the module `Horpo` contains the actual definition of the HORPO order.

```
Module Horpo (S: Types.Signature)
             (Prec: Precedence with Module S := S).
```

The definition of HORPO consist of five mutually recursive inductive definitions. The first one (`horpoArgs`) corresponds to $\succ\!\!\succ$ in Definition 6.1.

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

[1] Actually this fact has no impact on the development and the non-deterministic version could be used as well.

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

The second, crucial one (prehorpo), is the list of 6 clauses from Definition 6.1.

```
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,
       isApp M ->
       isApp N ->
       term (appHead M) = ^f ->
       term (appHead N) = ^g ->
       f >#> g ->
       M [>>] (appArgs N) ->
    M >-> N
| HMul:
    forall M N,
       isFunApp M ->
       isFunApp N ->
       term (appHead M) = term (appHead N) ->
       list2multiset (appArgs M) {>>} list2multiset (appArgs N) ->
    M >-> N
| HAppFlat:
    forall M N Ns,
      isFunApp N ->
      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)
```

The third one, `horpo`, is the actual HORPO relation ($\succ$). It extends `prehorpo` with requirement on types of terms (they have to be equivalent).

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

`horpoMul` is the extension of HORPO to multisets ($\succ_{mul}$).

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

And finally `horpoRC` is a reflexive closure of HORPO ($\succeq$).

```
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).
```

6.2. Proof of well-foundedness

In this section we present main result — proof of well-foundedness of the higher-order recursive path ordering introduced in Definition 6.1. The proof is due to Jouannaud and Rubio and was presented in [6]. It uses computability predicate method due to Tait and Girard (discussed in section 5.1) with some standard properties of computability (mentioned in section 5.2). In the remainder by computability we understand computability with respect to $\succ \cup \rightarrow_\beta$.

The first important lemma states that if all application arguments of functional application are computable then so is the whole application. First we are going to prove an auxiliary fact that will be used in the mentioned lemma.

Fact 6.1
If the following holds:

(1) $M \succ\!\!\succ \{N_1, \ldots N_k\}$

(2) all application arguments of M *are computable.*

(3) all N_i *are sub-terms of some term* N*.*

(4) if N' *is a sub-term of* N *and* $M \succ \cup \rightarrow_\beta N'$ *then* N' *is computable.*

then all N_i *are computable for* $i \in \{1, \ldots, k\}$*.*

Proof
For a given N_i we have two cases:

- $M \succ N_i$. Then N_i is a sub-term of N (using (3)) and $M \succ \cup \rightarrow_\beta N_i$ (because $M \succ N_i$), so we use (4) to show that N_i is computable.
- $M' \succeq N_i$ for M' being some application argument of M. If $M' = N_i$ then by (2) M' is computable and so is N_i. If, on the other hand, $M' \succ N_i$ then by property (2) of computability (Definition 5.1) N_i is computable because $M' \succ N_i$ and M' is computable (using (2)).

□

Now we can present the aforementioned lemma.

Lemma 6.2
If all $M_1, \ldots, M_k$ *are computable then so is* $f(M_1, \ldots, M_k)$*.*

Proof
The proof proceeds by well-founded induction on the pair $(f, \{\{M_1, \ldots, M_k\}\})$ ordered lexicographically by $\triangleright$ and the multiset extension of $\succ \cup \rightarrow_\beta$ restricted to finite multisets containing only computable terms (in the remaining of the proof we will denote this relation by $\ggg$). Computable terms are well-founded with respect to $\succ \cup \rightarrow_\beta$ (property (1) of computability) and so is its multiset extension (Theorem 3.14). That justifies the use of well-founded induction.

Now since $M = f(M_1, \ldots, M_k)$ is neutral we can use computability property (3). So we have $M \succ \cup \rightarrow_\beta N$ and it is left to show that N is computable. We show it by inner induction on the structure of N. There are few cases why $M \succ \cup \rightarrow_\beta N$ can hold and we will consider them in turn. The first case corresponds to beta-reduction step and the following ones to different clauses of $\succ$.

- Let $M \rightarrow_\beta N$. By Fact 4.5 we have that

  $$f(M_1, \ldots, M_i, \ldots M_k) \rightarrow_\beta f(M_1, \ldots M_i', \ldots M_k) \text{ with } M_i \rightarrow_\beta M_i'$$

  Since

  $$(f, \{\{M_1, \ldots, M_i, \ldots, M_k\}\}) \ggg (f, \{\{M_1, \ldots, M_i', \ldots, M_k\}\})$$

  and M_i' is computable by property (2) of computability we conclude that N is computable by the outer induction hypothesis.

- Let $M \succ N$ by case (1) of the HORPO definition, which means that $M_i \succeq N$ for some $i \in \{1, \ldots, k\}$. M_i is computable and so is N: if $M_i = N$ then it holds trivially and if $M_i \succ N$ then we show that by property (2) of computability.

- Let $M \succ N$ by case (2) of the HORPO definition. Then $N = g(N_1, \ldots N_l)$ with $f \triangleright g$. All N_i for $i \in \{1, \ldots, l\}$ are computable by Fact 6.1. Since

$$(f, \{\{M_1, \ldots, M_k\}\}) \ggg (g, \{\{N_1, \ldots, N_l\}\})$$

  we conclude that N is computable by the outer induction hypothesis.

- Let $M \succ N$ by case (3) of the HORPO definition. Then $N = f(N_1, \ldots N_l)$ and $\{\{M_1, \ldots, M_k\}\} >_{mul} \{\{N_1, \ldots, N_l\}\}$. We claim that all application arguments of N are computable. By Fact 3.5 for every $i \in \{1, \ldots, l\}$ we have some $j \in \{1, \ldots, k\}$ with either:

  - $N_i = M_j$ and then N_i is computable since so is M_j
  - or $M_j >^+_{mul} N_i$ and then N_i is computable by property (8) of computability.

  Now the fact that N is computable follows easily from the outer induction hypothesis.

- Let $M \succ N$ by case (4) of the HORPO definition. Then $@(N_1, \ldots, N_l)$ is some left-partial flattening of N and $M \succ\!\!\succ \{N_1, \ldots, N_l\}$. N is computable by property (9) of computability as all N_i for $i \in \{1, \ldots, l\}$ are computable by Fact 6.1.

- Cases (5) and (6) of the HORPO definition do not apply for a functional application on the left-hand side.

□

The next step is the definition of computable substitution followed by a proof that application of a computable substitution to an arbitrary term gives a computable term.

Definition 6.2 (Computable substitution)
We say that substitution $\gamma = [x_1/u_1, \ldots, x_n/u_n]$ is computable if u_i is computable for every $i \in \{1, \ldots, n\}$.

Lemma 6.3
$M\gamma$ is computable for every term M and every computable substitution γ.

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

- M is a variable $\Gamma \vdash x : \delta$. Then either $x\gamma = x$ in which case it is computable because every variable is computable. If, on the other hand, x is substituted then the result of substitution is the term $\Gamma \cdot \mathrm{Ran}(\gamma) \vdash t : \delta$ where $\gamma = [\ldots, x/\Delta \vdash t : \delta, \ldots]$. According to Lemma 4.4 the result can be represented equivalently as $\Gamma \cdot \mathrm{Ran}(\gamma) \cdot \Delta \vdash t : \delta$. Now this term is easily proven to be computable using computability property (7) and noticing that $\Delta \vdash t : \delta$ is computable as γ is a computable substitution.

- M is a function symbol f. Of course $f\gamma = f$ which is computable by Lemma 6.2.

- M is an abstraction $\lambda x : A.\ N$. Using property (5) of computability we are left to show that $N\gamma[x/P]$ is computable for every computable term P. We can see that $N\gamma[x/P] = N(\gamma \cup [x/P])$ because x may not occur in γ. Obviously $\gamma \cup [x/P]$ is a computable substitution and we can use induction hypothesis to conclude that $M\gamma$ is computable.

- M is an application $@(M_l, M_r)$. We use induction hypothesis to conclude that $M_l\gamma$ and $M_r\gamma$ are computable terms. Since $@(M_l, M_r)\gamma = @(M_l\gamma, M_r\gamma)$ by computability property (4) we conclude that $M\gamma$ is computable.

□

There are few differences between the proof presented above and the one from [6]. The first one is that we use simple induction on the structure of term whereas in mentioned paper more complex induction on the size of term without counting variables is used. The second difference is that we need only a trivial variant of Lemma 6.2, namely with empty list of application arguments. This comes from the fact that we use standard simply typed λ-calculus terms while in [6] a special form of function application (respecting the arity of function symbol) is used.

The main theorem, stating that the union of HORPO and beta-reduction on terms is well-founded, follows easily.

Theorem 6.4
The relation $\succ \cup \rightarrow_\beta$ is well-founded.

Proof
This follows directly from property (1) of computability and an applicatiton of Lemma 6.3 with the empty substitution (which is obviously computable).

□

The formalization of well-foundedness of HORPO can be found in the module `HorpoWf.v/HorpoWf`. First some general definitions are given, among them definition of relation of interest — `HB` (corresponding to $\succ \cup \rightarrow_\beta$) and of computable substitution — `CompSubst`.

```
Definition HB M N := M >> N \/ M -b-> N.
Definition HB_lt := transp Term HB.
Definition HB_mul_lt := MultisetLt HB.

Notation "X -HB-> Y" := (HB X Y) (at level 55).
Notation "X <-HB- Y" := (HB_lt X Y) (at level 55).

Definition Terms := list Term.
Definition CompHB := Computable HB.
Definition CompTerms (Ts: Terms) := AllComputable HB Ts.
Definition CompSubst (G: Subst) :=
   forall Q, In (Some Q) G -> CompHB Q.
```

Then a data-type representing list of computable terms is introduced as `WFterms`. It consist of a list of terms and a proof that all terms in the list are computable. It is needed in Lemma 6.2, where well-founded induction on a pair is used and a second component of a pair are computable terms ordered by $\succ \cup \rightarrow_\beta$ (their computability justifies the use of induction).

```
Definition WFterms := { Ts: Terms | CompTerms Ts }.
Definition WFterms_to_mul (Ts: WFterms) : Multiset :=
  list2multiset (proj1_sig Ts).
Coercion WFterms_to_mul: WFterms >-> Multiset.
Definition HB_WFterms_lt (M N: WFterms) := HB_mul_lt M N.
```

Then it has to be proven that the relation `HB_WFterms_lt`, which is a multiset extension of $\succ \cup \rightarrow_\beta$ restricted to computable terms, is well-founded. This is achieved in lemma `wf_MulLt_WFterms`. `compTerms` is the formalization of Lemma 6.1. Lemma 6.2 is split into three lemmas in Coq: `argsComp_appComp_aux` which deals with all cases from the proof, `argsComp_appComp_ind` which performs and justifies the well-founded induction and `argsComp_appComp` which proves the main goal. Lemma 6.3 is proven as `subst_comp` and Theorem 6.4 is stated as `horpo_beta_wf`.

Chapter 7

Conclusions and future work

We have presented the formalization of the proof of well-foundedness of HORPO in Coq. The complete proof presented in [6] has been formalized with all the notions and results the proof depends on (that includes part of the theory of simply-typed λ-calculus, multisets and the multiset extension of an order). The only hypothesis used are the computability properties but they are all standard and well-known.

Apart from simply verifying the correctness of proof presented by Jouannaud and Rubio [6] it can be regarded as contribution to Coq. To extend Coq with possibility of calculating using term-rewriting it is necessary to provide formal proofs that term rewriting systems being used are terminating. This work provides a base for that, that is a formal proof of correctness of the higher-order recursive path method.

On the other hand it can be used for the study of termination proved automatically. Several tools for proving termination of term rewriting systems exists and some of them use the (HO)RPO. Their output could be verified in Coq and for that purpose the proof of correctness of the HORPO method would be required.

There is a lot of possibilities to extend this work. We will list some of them.

- **Proving computability properties.**
  Computability properties are the only hypothesis in the development and proving them would make it complete thus it is probably the most obvious extension. But this would definitely require significant amount of work to complete.

- **Developing theory of term rewriting systems.**
  Only the HORPO relation and its well-foundedness are formalized. It is possible to extend it to include (part) of the theory of term rewriting systems. That means that first a definition of term rewriting system and term rewriting relation will have to be given. Also the proof of the HORPO being reduction order (monotonicity and stability under substitutions will be required) and conclusion that if for every rewrite rule $l \to r$ we have $l \succ r$ ($\succ$ standing for the HORPO relation) then the rewrite relation induced by given term rewriting system is terminating will be needed. Again this requires substantial amount of work.

- **Developing some decidability procedures.**
  For the time being only computational content of the development is the decision procedure which given environment and preterm computes its type and the type derivation or a proof that in that environment given preterm in not typable. When the term rewriting theory is specified it is easy to imagine procedures to perform a reduction step, to indicate whether given term is in normal form or even to compute normal form of a

term. The ultimate goal would be the decidability of the order (that is given terms l and r to decide whether $l \succ r$ or not, where $\succ$ stands for the HORPO) but that problem has not yet been investigated even by the authors of the HORPO as stated in [7].

- **Adaptation of the proof to other frameworks.**
  In our development we work with plain syntactic pattern matching and then prove termination of union of the HORPO relation and the β-reduction relation of simply typed λ-calculus. Another approaches are being used as well. For example the higher-order rewriting systems introduced by Nipkow define rewriting modulo $\beta\eta$ of simply typed λ-calculus and work with terms in η-long normal form. The adaptation of the proof presented in this document to that approaches is possible as mentioned in [6] and further explained in [13] (where the variant of rewriting used in this work is referred as AFS and the variant introduced by Nipkow as HRS). It would be interesting to extend the development to those different variants of higher-order rewriting.

- **Proving the well-foundedness of stronger versions of the HORPO**
  Finally the formalized version of the HORPO is the simplest one presented in [6]. It was then extended in the same document and even further in [7]. The last version is much more powerful. It also includes the β-reduction relation of simply typed λ-calculus.

Bibliography

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

[2] B. Barras and B. Werner. Coq in Coq. Soumis, 1997.

[3] S. 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.

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

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

[6] 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.

[7] 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`, 2003.

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

[9] F. Leclerc. Termination proof of term rewriting systems with the multiset path ordering: A complete development in the system Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications (TLCA'95)*, volume 902 of *LNCS*, pages 312–327, Edinburgh, UK, April 1995. Springer.

[10] C. Murthy. *Extracting constructive content from classical proofs.* PhD thesis, Cornell University, New York, USA, 1990.

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

[12] H. Persson. *Type Theory and the Integrated Logic of Programs.* PhD thesis, Göteborg University, Göteborg, Sweden, May 1999.

[13] 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.

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