Well-foundedness of RPO in Coq

Master of Science thesis by:

Nicole de Kleijn

August 2003

Supervisor: Femke van Raamsdonk
Second reader: Evert Wattel

Free University Amsterdam
Faculty of Sciences
Department of Mathematics

Contents

Preface

This thesis is the final work for the Master of Science degree in Mathematics. The thesis is made in cooperation with the Theoretical Computer Science group of the Faculty of Sciences of the Free University in Amsterdam. The supervisor of this thesis is therefore from the department of Computer Science. The second reader is from the department of Mathematics.

I became first interested in theoretical computer science during the courses 'Applied Logic' and 'Coding and Complexity'. During the course applied logic I had my first experience with the proof checker Coq. It was pleasant and I was interested in knowing more about Coq. During the course coding and complexity I first learned about undecidability and Turing machines. Because of these courses I decided to do my Master's thesis in theoretical computer science.

The subject of this thesis is to formalize some theory of term rewriting systems in the proof checker Coq. Looking back after finishing the thesis, I am content with the result, the main goal is reached. From January 2003 till August 2003 I worked with a lot of pleasure on the Coq files and the thesis. Sometimes formalizing the theory seemed a neverending sequence of problems, but fortunately it turned out to be well-founded.

I want to thank Femke for her supervision to my project. Our cooperation was pleasant and inspiring. Her enthusiastic and fast reactions to my questions were always helping me out. Also thanks to Freek, who gave me some hints via Femke. I really enjoyed working with her.

Thanks to Evert Wattel for his interest in my work, reading my thesis and listening to my talk about my thesis.

Finally thanks to my parents, my brothers and Arnold, who supported me during my whole study. They always put confidence in my choices and helped me in making choices. I hope they all will try to read my thesis, to get an idea of what I have done the past semester.

Chapter 1

Introduction

The subject of this thesis is the formalization of a small part of the theory of term rewriting systems. To be more precise, the aim is to formalize some theory about the recursive path ordering. The formalization is done with the proof assistant Coq. This chapter will first provide some background information about term rewriting systems and Coq. We consider some related work, and the goals of this work are presented. Finally an overview of the thesis is given.

1.1 Background

1.1.1 Term rewriting systems

A term rewriting system (TRS) consists of a signature and a set of rewrite rules. The rewrite rules work on first-order terms, which are built from the signature and a set of variables. The set of terms is denoted as $\mathcal{T}(\Sigma, X)$, where X is the set of variables. The signature is often denoted as Σ, the set of rewrite rules is denoted as R and a TRS is denoted as (Σ, R). In this thesis the signature can be finite or infinite, but the set of rewrite rules is always finite. Two properties which are central in the theory of TRSs are confluence and termination. A TRS is confluent if co-initial diverging rewrite sequences can be joined. A TRS is terminating if all rewrite sequences are finite. This thesis is concerned with on termination.

Proving termination of a given TRS (Σ, R) is in general undecidable. Many methods and techniques to prove termination of TRSs have been developed. A well-known way to prove termination of a TRS is to design an ordering $>$ on terms that satisfies the following two properties. First this ordering has to ensure that for all terms s and t, if the term s is rewritten to the term t then $s > t$. If the ordering descends to infinity, termination cannot be proven. Second the ordering has to satisfy another property, called well-foundedness. Well-foundedness of an ordering on a set provides that every descending sequence in this ordering is finite. If well-foundedness of $>$ is satisfied, termination of (Σ, R) follows.

An ordering on terms which satisfies the above two conditions is the recursive path order (RPO). This ordering is roughly speaking defined by extending a well-founded ordering on function symbols in a recursive way to an ordering on the set of terms. Using RPO on the terms, termination of a TRS can be proven. That RPO indeed is an ordering, and well-foundedness of RPO are the main topics of this thesis.

1.1.2 Formalization, proof checking and Coq

The subject of this thesis is to formalize the definition of RPO, and RPO being an ordering, and RPO being well-founded. Formalizing theory requires a detailed understanding of this theory. Using the proof-assistant Coq, proofs can be checked on correctness. For this thesis version 7.3.1

of Coq is used. The proof assistant consists of two parts: the proof development system and the proof checker. In the proof development system the formalized theory is defined and when neccessary proven. By running the proof checker on this code in the proof development system the formalized theory and proofs are checked on correctness. Coq is based on type theory, which makes the base of Coq depend on the Curry-Howard-De Bruijn isomorphism between λ-calculus and logic. Hence proof checking in Coq is actually type checking. The formal language which is the kernel of Coq is the Calculus of Inductive Constructions, denoted as Cic. The Cic is a formulation of type theory including the possibility of inductive constructions. The Calculus of Constructions is introduced in [CH88]. In [CPM89] inductive definitions are introduced.

Coq contains three universes where objects can come from. This universes are also called sorts.

`Prop` This is the universe of logical propositions. If `A:Prop` then `A` denotes the class of terms representing a proof of the proposition `A`. `Prop` contains `True` and `False`.

`Set` This is the universe of program types or specifications. For instance the datatypes booleans and natural numbers are in `Set`.

`Type` This is the universe of both `Set` and `Prop`. This gives `Set:Type` and `Prop:Type`.

In Coq inductive types are extensively used. An inductive type is built from constructors. Application of this constructors gives the set which is represented by the type. The inductive type is minimal in the sense that the number of constructors is minimal. As an example the definition of the natural numbers is given. In Coq the natural numbers are defined as follows:

```
Inductive nat: Set  :=  O : nat
                      | S : nat->nat.
```

This definition is read as follows: `nat` is inductively defined and `nat` represents an element of type Set. There are two constructors `O` and `S`. The constructor `O` represents 0 and is indeed a natural number. `S` represents the successor function, which gives from a natural number it's successor. With these two constructors all natural numbers can be formed. The set `nat` contains exactly all finite expressions built from `O` and`S`.

The finite lists of natural numbers are also defined inductively in Coq:

```
Inductive natlist: Set :=
  nil  : natlist
| cons : nat->natlist->natlist.
```

This definition is conform the usual definition of a list. It reads as follows: the empty list is a `natlist` and joining a `nat` (head) and a `natlist` (tail) gives a new `natlist`. In the standard library of Coq, which is available at `http://pauillac.inria.fr/coq/`, polymorphic lists are already formalized in the module `Coq.Lists.PolyList`.

Recursive definitions are also often used in Coq, so an example of an recursive definition is given too. The function `plus` which adds two natural numbers is defined as follows:

```
Fixpoint plus [m,n:nat]: nat :=
Cases n of
  O     => m
| (S p) => (S (plus p m))
end.
```

This definition takes as input two natural numbers m and n and returns a thing of type `nat`. It distinguishes the two cases of n to determine the sum of m and n.
More information about Coq can be found in [Coq03].

1.2 Related work

RPO was first introduced by Dershowitz in [Der82]. Dershowitz defined RPO using the multiset ordering. RPO being a strict partial order is proved during the proof that RPO is a simplification ordering. Well-foundedness of RPO is proved using Kruskal's tree theorem. Because this proof is a classical proof, it isn't very usefull to formalize. The original RPO is sometimes called the multiset path ordering. More information about rewriting and termination is found in [Der87], [DJ90], [BN98] and [Ter03].

After its initial introduction some variants of RPO appeared. In [KL80], RPO using the lexicographic order is introduced under the name lexicographic path order. Combining the lexicographic path order and the multiset path order gave the recursive path order with status. In [Fer95] a criterion is given for path orders on terms, that in combination with the subterm property is shown to guarantee well-foundedness. All mentioned path orders satisfy this criterion.

In this thesis the most original form of RPO is chosen, the one using the multiset ordering. This multiset ordering has to be a strict partial order and well-founded. A constructive proof of the multiset ordering being well-founded is found in [Nip98].

A proof of RPO being well-founded, not using Kruskal's tree theorem, is implicitly given in [JR99]. This paper contains a constructive proof ofthe well-foundedness of the higher-order recursive path ordering (HORPO). HORPO is an extension of RPO to higher order terms. The explicit proof for the first order case is given in the appendix of [Raa01]. This proof is used in this thesis.

The proof in [JR99] uses the lexicographic ordering on the product of tripels. This lexicographic ordering has to be a strict order and well-founded as well. More about this is found in [Pau86] and [BN98].

Leclerc [Lec95] presents a formalization of RPO in Coq. The formalization is not presented in all detail because it takes 250 pages. The focus is on giving upper bounds for descending sequences in RPO. The formalization we present here seems to be quite different from the one by Leclerc. For instance, the formalization of the data type terms is different, and the formalization of RPO is different. Our formalization is shorter.

Jean Goubault-Larrecq gave a proof of well-foundedness of a certain binary relation in [GL01]. The proof doesn't depend on terms and can also be applied to graphs, infinite terms and other domains. Goubault-Larrecq showed that the theorem about the well-foundedness of certain binary relations subsumes the theory of Ferreira, see [Fer95]. The consequence of this is that the theory subsumes RPO as well. The general theory is checked in Coq. Showing in Coq that RPO fulfills the conditions of the theory, gives implicit a proof of RPO being well-founded. The purpose of the present project is to give a explicit proof of RPO being well-founded, following the proof of the long version of [Raa01] exactly.

Henrik Persson proved the well-foundedness of RPR and formalized it in the Agda type checker in [Per03]. RPR is a weakening of RPO, because it being an order is no longer required. The proof of RPR being well-founded is similar to the proof in [JR99] and [Raa01].

1.3 Goal

The goal of this project is to formalize several parts of the theory of RPO. In the next list the main goals are summarized:

Formalize the original definition of RPO due to Dershowitz in Coq.

Formalize a proof of RPO being a strict partial order in Coq.

Formalize a proof of RPO being well-founded in Coq using [JR99].

Because the proof of [JR99] uses the lexicographic order and because in the definition of RPO by Dershowitz the multisets are used, the following goals were added during the formalization process:

Formalize the lexicographic order of the independent product of tripels and prove it's a strict partial order and it's well-founded.

Formalize the multisets, the multiset order and prove it's a strict partial order and it's well-founded.

1.4 Overview

While reading this thesis try to place the Coq files from the Appendix next to the thesis. The Coq files are also online via `www.cs.vu.nl/~tcs`. The correspondence between the Coq files and the thesis will best shown in this way. For every chapter there are one or more Coq files which are found in one of the appendices. Where possible the examples from the thesis are implemented in the Coq files too.

A summary of the different chapters is given next. In Chapter 3 the terms are introduced and some more detailed theory about term rewriting systems is given to understand why RPO is needed. Of this chapter only the part about terms is formalized. The Coq file `Terms.v` is found in Appendix B.

In Chapter 4 orderings and its properties are introduced. The Coq file `Ordering.v` is found in Appendix C. In Chapter 5 the notion of well-foundedness is explained, well-founded induction and well-founded part induction are introduced. The Coq file `Wellfounded.v` is found in Appendix D.

In Chapter 5 the lexicographic order on a product is introduced. The proofs of this being a strict partial order and being well-founded are given for the product of two sets and the product of three sets. The Coq files `Lex2.v` and `Lex3.v` are found in Appendices E and F.

Multisets and the multiset ordering are introduced in Chapter 7. In Coq only multiset reduction is formalized. A consequence of this is that the multiset ordering being a strict partial order isn't formalized either. Well-foundedness of this multiset ordering is formalized and found in the Coq file `Multisets.v` in Appendix G.

In Chapter 8 the recursive path order is finally introduced. RPO being a strict partial order and being a reduction order is only proven on paper. One reason for this is that only multiset reduction was formalized, so RPO being a strict partial order couldn't be formalized. The file `RPO.v` is found in Appendix H.

Chapter 2

Term rewriting systems

2.1 Definition of the set of terms

The Coq code corresponding to this Section is found in Appendix B. Terms are built from a signature and a set of variables. First the notion of a signature is defined.

Definition 2.1 A *signature* Σ is a set of function symbols. All these function symbols carry a non-negative integer n with them, which is called the *arity* of the function symbol.

Some remarks concerning this definition are to be made. When a function symbol f has arity n it means that f will expect n arguments. $\Sigma^{(n)}$ is the notation for all function symbols with arity n. The elements of $\Sigma^{(0)}$ are often called constant symbols. In this definition of the signature all function symbols have a fixed arity. But a signature with function symbols with varyadic arity is also possible. The signature is then defined as follows:

Definition 2.2 A *signature* Σ is a set of function symbols. All these function symbols carry a set of a non-negative integers with them, which are the possible arities of the function symbol. All arities are finite, but a function symbol can have infinitely many arities.

In both 2.1 and 2.2 the signature can be infinite. Remark that a signature with varyadic arity can be made into a signature with fixed arity by labelling the function symbols with an extra label, which denotes the arity.

Example 2.3 An example of a signature with fixed arities is $\Sigma = \{0, S, M, A\}$, with 0 a constant symbol, S of arity 1 and A and M of arity 2. A semantic interpretation of these function symbols is 0 as zero, S as the successor function, and A and M as the addition and multiplication functions.

Example 2.4 Take as the signature the set $\mathbb{N}$ and suppose that every element of $\mathbb{N}$ can have all possible arities. This signature has varyadic arity. This means that for instance 0 can have arity 0, 1, 2 and so on.

The difficulty of defining a signature with fixed arity in Coq was that it's natural to define the signature as an function from the set of function symbols to the natural numbers. The problem in Coq was to guarantee that the right arity was with the right function symbol. And an even bigger problem was to translate the arity to the number of arguments a function symbol expects. The only way to ensure the above problems don't occur to give every function symbol with it's arity explicitly. This restricts the signature to be finite and always explicitly given. In Coq the signature is choosen to be the set `nat`, because `nat` already exists in Coq. All elements of `nat` can have all possible arities, as in Example 2.4. The restriction that the signature has to be finite is gone. A consequence of the choice is that RPO will only be formalized for the signature being equal to `nat`.

Terms are defined as follows:

Definition 2.5 Let Σ be a signature and X a set of variables with $\Sigma \cap X = \emptyset$. Then the set of terms $\mathcal{T}(\Sigma, X)$ over Σ and X is inductively defined as follows:

1. every x with $x \in X$ is a term,
2. for all $n \geq 0$, for all $f \in \Sigma^{(n)}$ and for all $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, X)$, we have $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, X)$.

In words this definition means that first every variable is a term. Second, the application of a function symbol to a list of terms gives a new term. If c is a constant symbol, write c instead of $c()$.

Example 2.6 Assume a set of variables X the elements of which are denoted as x,y,z,.... With the signature Σ of Example 2.3 the following expressions are examples of terms:

$$
\begin{array}{l}
0 \\
x \\
S(0) \\
A(S(x), y) \\
M(S(x), A(S(0), S(y))) \\
S(M(0, S(y))) \\
M(A(x, x), A(x, y))
\end{array}
$$

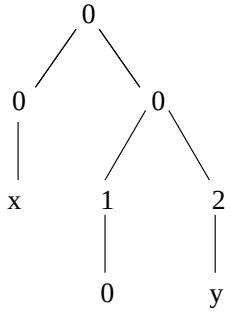
Some expressions which are no terms are the following:

$S(x, x)$	(arity $S = 1$)
$M(0)$	(arity $M = 2$)
$A(x, S(0)$	(missing parenthesis)
M	(M expects 2 arguments)
$D(x)$	(D is not a function symbol)

Example 2.7 Assume a set of variables X the elements of which are denoted as x,y,z,.... With the signature $\mathbb{N}$ of Example 2.4 the following expressions are terms:

$$
\begin{array}{l}
0 \\
x \\
0(0) \\
0(0, 0) \\
0(0(0), 0) \\
0(0(x), 0(1(0), 2(y))) \\
0(0(5, 5(y)))
\end{array}
$$

Terms which consists only of function symbols and therefore don't contain variables are called *ground terms* or *closed terms*. Terms which may contain occurrences of variables are called *open terms*. Examples of ground terms are the first and third term of Example 2.6 and for instance $M(S(S(0)), S(0))$. In better understanding the structure of terms, a term is usually denoted as a tree, with the outher function symbol as root and the variables and constants as leaves. The term tree of the term $0(0(x), 0(1(0), 2(y)))$ of Example 2.7 is as follows:



In Coq, the set of ground terms over the signature $\mathbb{N}$ is defined by means of mutual recursion. The natural number are represented by `nat (:Set)`.

```
Inductive term : Set :=
  buildterm : nat -> termlist -> term
with
  termlist : Set :=
    nilterm : termlist |
    consterm : term -> termlist -> termlist.
```

It is also possible to define the set of open terms over the signature in $\mathbb{N}$ in Coq:

```
Variable var:Set.

Inductive term : Set :=
  varterm : var -> term |
  buildterm : nat -> termlist -> term
with
  termlist : Set :=
    nilterm : termlist |
    consterm : term -> termlist -> termlist.
```

Both definitions are mutual inductive definitions, because `term` depends on `termlist` and vice versa. The definition of ground terms is read as follows: application of a function symbol and a list of ground terms gives a new ground term. This is exactly the definition of ground terms.

In the definition of open terms a set `var` is added. A consequence of this addition is that the set `var` has to be identified in almost every definition, which makes the Coq code less transparent. Choosing the set `nat` for the set of variables is not allowed, because the signature and the set of variables have to be disjunct. Because the Coq code for open terms is less transparent than the one for ground terms, the rest of this thesis only considers Coq code for ground terms. The Coq code can be found in Appendix B.

Applying induction on terms is not trivial here. If a property has to be shown for a `term`, there has to be an equivalent property for the corresponding `termlist`. Induction on a `term` and induction on a `termlist` have to be combined to one induction principle. In Coq this is done by `Scheme` as follows:

```
Scheme
  term_induction := Induction for term Sort Prop
with
  termlist_induction := Induction for termlist Sort Prop.
```

To see if this induction scheme really is the induction principle which is needed, check this in Coq.

```
Check term_induction.

term_induction
     : (P:(term->Prop); P0:(termlist->Prop))
       ((n:nat; t:termlist)(P0 t)->(P (buildterm n t)))
       ->(P0 nilterm)
       ->((t:term)(P t)->(t0:termlist)(P0 t0)->(P0 (consterm t t0)))
       ->(t:term)(P t)
```

The scheme is read as follows: The property on `term` which is to be proven is P and the equivalent property on `termlist` is P0. First if P0 holds for a termlist t, then P holds for the term built from a natural number n and t. Second induction on the termlist gives that P0 has to

hold for the empty termlist and P0 has to hold for the list built from a term t and a termlist $t0$, where P holds for t and P0 holds for $t0$. From this follows that P holds for all terms.

There are several features of terms that are interesting. A function that gives the root of a term and a function that gives the proper subterms of a term are formalized in Coq as `root` and `subterms`. The length of a term is defined as follows:

Definition 2.8 The length of a term t is the number of occurrences of function symbols and variables in t. Notation: $|t|$. The inductive definition of $|t|$ is as follows:

1. $|x| = |c| = 1$, with $x \in X$ and $c \in \Sigma^{(0)}$,
2. $|f(t_1, \ldots, t_m)| = |t_1| + \ldots + |t_m| + 1$, for $m \geq 1$ and $f \in \Sigma^{(m)}$.

Example 2.9 The lengths of the terms from Example 2.6 are as follows:

term	*length*
0	1
x	1
$S(0)$	2
$A(S(x), y)$	4
$M(S(x), A(S(0), S(y)))$	8
$S(M(0, S(y)))$	5
$M(A(x, x), A(x, y))$	7

Example 2.10 The lengths of the terms from Example 2.7 are as follows:

term	*length*
0	1
x	1
$0(0)$	2
$0(0, 0)$	3
$0(0(0), 0)$	4
$0(0(x), 0(1(0), 2(y)))$	8
$0(0(5, 5(y)))$	5

In Coq, the length of a ground term is recursively defined with a mutual recursive definition. The length of a constant ground term is one and the length of a ground term build from a head and a tail is equal to length(head) + length(tail), where the head is a ground term and the tail is a list of ground terms.

```
Fixpoint length_term [t:term] :nat :=

Cases t of
  (buildterm p q) => (S (length_termlist q)) end
with
  length_termlist [q:termlist]: nat:=
     Cases q of
     nilterm => 0
|    (consterm k l) => (plus (length_term k) (length_termlist l)) end.
```

Lists are already formalized in Coq. They can be found in the library in module `Coq.Lists.PolyList`. The definition of polymorphic lists over A is as follows:

```
Inductive list : Set :=
  nil : list
| cons : A -> list -> list.
```

We will need to convert terms in `termlist` (from the definition of terms) to `(list term)` and vice versa. These conversions are implemented in `termlist_to_list` and `list_to_termlist`. An element is in a termlist if it's in the list of terms which arises by converting the termlist with `termlist_to_list`. This is formalized in `in_termlist`.

2.2 The rewrite relation

As said before, a term rewriting system (TRS) consists of a signature and a set of rewrite rules. From the signature the first-order terms are built as explained in Section 2.1. The rules operate on terms. Rules are denoted as $l \to r$, where l stand for the left-hand side and r for the right-hand side. The rules are subject to the following restrictions:

The left-hand side is not a variable

Every variable in r also occurs in l.

In this thesis the set of rules is always finite. A TRS is denoted as (Σ, R), where Σ is the signature and R is the set of rules. A TRS with rules which only consists of ground terms is called a ground TRS. Remark that this is not the same as the restriction of a normal TRS to ground terms only, because then still the rules can contain variables as well.

In this way we obtain rewrite steps. First an *instance* of a rule ρ is obtained by applying a substitution σ to a rule ρ. $l^\sigma \to r^\sigma$ is obtained here. l^σ is called an *redex* and r^σ its *contractum*. A term can contain zero, one or more redexes. By contracting one of these redexes in the surronding context, a rewrite step is made. The formal definition is as follows:

Definition 2.11 A rewrite step which comes from a rewrite rule $\rho : l \to r$ consists of contracting a redex in an arbitrary context:

$$C[l^\sigma] \to_\rho C[r^\sigma]$$

$\to_\rho$ is called a one-step reduction relation generated by ρ. The reduction relation of a TRS is the union of $\to_\rho$ for all reduction rules ρ.

A reduction sequence is a chain of rewrite steps, for instance $s_0 \to s_1 \to s_2 \to \ldots$.

2.3 Termination

Termination of a TRS is defined as follows:

Definition 2.12 An term s from $\mathcal{T}(\Sigma, X)$ is called *terminating* if every reduction sequence starting with s is finite. A reduction relation $\to$ on $\mathcal{T}(\Sigma, X)$ is terminating if all elements of $\mathcal{T}(\Sigma, X)$ are terminating with respect to the relation $\to$. A TRS is terminating if the reduction relation of the TRS is terminating.

In general, proving termination of a TRS is undecidable. This means that there is no algorithm that decides whether a given TRS terminates or not. Undecidability of termination is for instance proved by using the undecidability of the uniform halting problem for Turing machines. A translation from proving termination to the uniform halting problem is made. Thereby proving termination is undecidable too. This proof is found in [Ter03] and [BN98].

However, termination of a TRS can be proved under certain constraints. There are three sorts of methods for proving termination: semantical methods, syntactical methods and transformational methods. Here we use a syntactical method. Syntactical methods are based on orders on terms. The idea is that an order can be used to prove termination if it's well-founded and if for every $s \to t$ we have $s > t$. In this thesis the order is a strict partial order.

Theorem 2.13 A TRS (Σ, R) is terminating if and only if there exists a well-founded strict partial order $>$ on $\mathcal{T}(\Sigma, X)$ such that for every $s, t \in \mathcal{T}(\Sigma, X)$ with $s \to_R t$ it holds that $s > t$.

Proof
$\Rightarrow$ If the TRS (Σ, R) is terminating, take for the strict partial order the transitive closure of $\to$.

$\Leftarrow$ If $>$ is well-founded, it follows that every reduction sequence in (Σ, R) has to be finite and thus that (Σ, R) is terminating.

Theorem 2.13 isn't yet very usefull, because usually there are infinitely many $s, t \in \mathcal{T}(\Sigma, X)$ with $s \to_R t$. To overcome this problem, the well-founded strict partial order has to be closed under substitution and closed under contexts. A consequence of this is that checking $l > r$ for all rules sufficies to prove termination. A strict partial order which is closed under substitutions and contexts is called a *reduction order.*

Definition 2.14 A *reduction order* on $\mathcal{T}(\Sigma, X)$ is a well-founded strict partial order which is:

1. closed under substitutions: if $s > t$ and σ is a substitution, then $s^\sigma > t^\sigma$,
2. closed under contexts: if $s > t$ and C is a context, then $C[s] > C[t]$.

For reduction orders we have the following result:

Theorem 2.15 A TRS (Σ, R) is terminating if and only if there exists a reduction order with $l > r$ for every rule $l \to r$ in R.

Proof
$\Rightarrow$ If the TRS (Σ, R) is terminating, take for the strict partial order the transitive closure of $\to$. This strict partial order $>$ is closed under substitutions and contexts, because $\to$ is.

$\Leftarrow$ If $>$ is a reduction order, and $l > r$ holds, then from the closure under substitutions and contexts follows that $C[l^\sigma] > C[r^\sigma]$ for all arbitrary contexts C and substitutions σ. That is if $s \to t$ then $s > t$. From the well-foundedness of $>$ it follows that (Σ, R) is terminating.

In Chapter 7 it is shown that RPO is a strict partial order, that it is a reduction order, and that it is well-founded. This yields the following method of proving termination of a term rewriting system: define a well-founded ordering on the function symbols and show that in its extension to the recursive path order it holds that $l > r$ for every rewrite rule $l \to r$.

More on TRSs can be found in [Ter03] and [BN98].

Chapter 3

Orderings

3.1 Basic definitions

Given a set S, a relation $R \subseteq S \times S$ relates elements of S by putting them into pairs. When $s, t \in S$ are related we denote this as sRt. Some properties of relations on a set are given in the following definition:

Definition 3.1 Given a set S and a relation R, define the following:

R is reflexive	$\Leftrightarrow$	$\forall x \in S : xRx$
R is irreflexive	$\Leftrightarrow$	$\forall x \in S : \neg(xRx)$
R is transitive	$\Leftrightarrow$	$\forall x, y, z \in S : (xRy \wedge yRz) \Rightarrow xRz$
R is symmetric	$\Leftrightarrow$	$\forall x, y \in S : xRy \Rightarrow yRx$
R is antisymmetric	$\Leftrightarrow$	$\forall x, y \in S : xRy \wedge yRx \Rightarrow x = y.$
R is an equivalence relation	$\Leftrightarrow$	(R is reflexive) $\wedge$ (R is transitive) $\wedge$ (R is symmetric)

Given a set and a relation, a natural question is whether the relation is an ordering on the elements of the set. Several sorts of orderings are defined in Section 3.2.

3.2 Orderings

Definition 3.2

- A *quasi order* $\geq$ on a set S is a relation on S which is reflexive and transitive.
- A *strict order* $>$ on a set S is a relation on S which is irreflexive and transitive.
- An *order* $\geq$ on a set S is a relation on S which is reflexive, transitive and antisymmetric.

The quasi order is added to complete the list of orderings, it will not be used in the rest of this thesis. All three orderings from Definition 3.2 are either partial or total. In a total ordering all elements of S are comparable and in a partial ordering this isn't necessarily the case. These two properties of orderings are defined as follows:

Definition 3.3

- An ordering $\geq$ on a set S is total if and only if $\forall x, y \in S : x > y \vee x = y \vee y > x$.
- An ordering $\geq$ on a set S is partial if it isn't total.

The equality used in this definition is syntactical equality. A strict order $>$ on S is total if and only if $\forall x, y \in S : x > y \vee y > x$. A strict order is partial if it isn't total. In a partial order there are at least two elements which are incomparable with respect to $>$.

To avoid confusion concerning the combination of Definitions from 3.2 and 3.3 we will explain our terminology for the rest of the thesis here. With a strict partial order we mean a strict order which is partial. With a partial order we mean an order which is partial. With a strict total order we mean a strict order which is total. And finally with a total order we mean an order which is total. The ordering which shall be used most in this project is a strict partial order.

In Coq, all above definitions are formalized; the Coq code is found in Appendix C. Sometimes it's useful to extend a relation in a certain way. An example of this is to add equality to a strict relation. In this way the relation is made reflexive. Another example is to extend a relation in such a way that it becomes transitive. By adding new related pairs to the relation, we get a new relation which satisfies the given properties.

The property the extended relation has to satisfy will be called P. The P closure of a relation R is the least set with property P which contains R. Two closures which are interesting for this project are the reflexive closure and the transitive closure.

Example 3.4 The set of natural numbers $\mathbb{N}$ with the usual order $\geq$ is a total order.

Example 3.5 Consider the set $X = \{0, S, A, M\}$.

1. The relation $>$ defined by: $M > 0$, $M > S$, $M > A$, $A > S$, is a strict partial order. This relation is a strict order because it's irreflexive and transitive. The order is not total, because for instance the elements 0 and S are not related. This is formalized in `R1strictpartialorder` in Appendix C.

2. The relation $>$ defined by: $M > 0$, $M > S$, $M > A$, $A > S$, $M > M$, $A > A$, $S > S$, $0 > 0$, is a partial order. This relation is an order because it's reflexive, transitive and antisymmetric. The order is partial, because for instance the elements 0 and S are not related. Note this relation is exactly the reflexive closure of the order from item 1. See `R2partialorder` in Appendix C for a formalization.

3. Ordering the elements of X as $M > A$, $M > S$, $M > 0$, $A > S$, $A > 0$ and $S > 0$ gives a strict total order. This order is total because every possible pair with distinct elements is in the relation. See `R3stricttotalorder` in Appendix C for a formalization.

4. Ordering the elements of X as $M > A$, $M > S$, $M > 0$, $A > S$, $A > 0$, $S > 0$ and $M > M$, $A > A$, $S > S$, $0 > 0$ gives a total order. Note this is exactly the reflexive closure of the order from item 3. See `R4totalorder` in Appendix C for a formalization.

Chapter 4

Well-foundedness

4.1 Definition of well-foundedness

Well-foundedness is a property of an ordering on a set. Before defining the notion of well-foundedness, first accessibility and well-founded part are defined. Informally, an element $a \in A$ is called accessible if it can be proven that every descending sequence starting with a is finite. The well-founded part of a set A with respect to $>$ is exactly the subset of accessible elements of A. An element $a' \in A$ is called a *successor* of $a \in A$ if $a > a'$. The formal definition of accessibility is as follows:

Definition 4.1 Given a set A and a relation $>$. Denote the well-founded part of A with W_A. An element of A is called accessible if it can be proven that all successor elements are accessible i.e., all successor elements are in the well-founded part of A. Inductively this gives:

$$\frac{\forall y \in A : x > y \Rightarrow y \in W_A}{x \in W_A}$$

In this scheme the line is the same as an implication symbol. The well-founded part of A ordered by $>$ is in general denoted by W_A. If A is clear from the context or irrelevant, sometimes the subscript A is omitted and W instead of W_A is written.

Remark 4.2 Note that from this inductive definition it follows that if an element a from A is accessible, then all its successor elements are accessible too.

If all elements of A are in the well-founded part of A, the ordering $>$ on the set A is called well-founded. The formal and equivalent definition is as follows:

Definition 4.3 An ordering $>$ on a set A is called well-founded if all elements of A are accessible in A.

Example 4.4 The strict order $>$ on the set $\mathbb{N}$ of natural numbers is well-founded. For every $n \in \mathbb{N}$, every descending sequence starting with n is finite. The strict order $>$ on the set $\mathbb{Z}$ of integers is not well-founded. Clearly for every $z \in \mathbb{Z}$ every descending sequence starting with z can be extended to an infinite descending sequence.

In Coq well-foundedness is already defined in the library. The definition of accessibility and well-foundedness are found in the module `Coq.Init.Wf`. But in this definition the relation R is thought of as a 'less than' relation. In this thesis the relation R is thought of as a 'greater than' relation. There was a choice between using the definition of well-foundedness from the library and orienting all relations opposite as in the thesis or making a new definition of well-foundedness by only orienting the relation in this definition as a 'greater than' relation. The last option is chosen to preserve transparency in the Coq code, which is found in Appendix D. Accessibility is denoted with `Access` and well-foundedness is denoted with `wellfounded`, they are formalized as:

```
Variable A : Set.

Inductive Access : A -> Prop
    := Access_intro : (x:A)((y:A)(gtA x y)->(Access y))->(Access x).

Definition wellfounded := (a:A)(Access a).
```

Note that the definitions of well-foundedness and accessibility are easily recognized in this Coq code. An element from A is accessible if it can be proven that all successor elements are accessible. And a relation is well-founded if all elements from A are accessible. The Coq code is found in `Wellfounded.v` in Appendix D.

The fact that from x is accessible follows that all y with $x > y$ are accesible is formalized in `Access_inv`:

```
Lemma Access_inv :
(x:A)(Access x) -> (y:A)(gtA x y) -> (Access y).
```

4.2 Induction principles

In the proofs of orderings being well-founded two concepts of well-founded induction are used. Before introducing these concepts of induction it's useful to remind the usual induction principle. This principle is founded on the natural numbers. The property $P(n)$ holds for all $n \in \mathbb{N}$, if it can be proved that $P(0)$ holds (base case) and it can be proved that $P(n')$ holds under the assumption that $P(n'-1)$ holds for $n > 0$ (induction step). This principle works because $\mathbb{N}$ is well-founded, i.e. there is no infinitely descending sequence in $\mathbb{N}$.

From this last remark two concepts of induction are made. The first concept is the principle of well-founded induction. This principle can only be used if the relation $>$ is well-founded. Without this the induction argument will fail. Define well-founded induction as:

Definition 4.5 Given a set A and a well-founded strict partial order $>$ on A. Suppose it can be proven that for all x in A, $P(x)$ holds under the assumption that for all y in A holds that if $x > y$, then $P(y)$. From this follows that $P(x)$ holds for all x in A. Summarized in a scheme:

$$\frac{\forall x \in A(\forall y \in A : x > y \Rightarrow P(y)) \Rightarrow P(x)}{\forall x \in A : P(x)}$$

The induction hypothesis for well-founded induction is $\forall y \in A : x > y \Rightarrow P(y)$. At first sight it seems that well-founded induction has no base case, but this is hidden in the induction hypothesis. Because $>$ is well-founded there are elements for which there are no smaller elements. Call these elements minimal elements. For the base case these elements are to be considered. Take an arbitrary minimal element x. For this x has to be proven that $P(x)$ holds. But because there are no elements smaller than x, the statement $\forall y \in A : x > y \Rightarrow P(y)$ is trivially true. This explains a base case doesn't need to be considered in well-founded induction.

In the library of Coq, the principle of well-founded induction is found in the module `Coq.Init.Wf`. We use an adapted version of the theorem `well_founded_ind` from `Coq.Init.Wf`, where we use a 'greater than' relation, and the proof is slighly different. Well-founded induction is formalized in `wf_ind` as follows:

```
Theorem wf_ind :
 (P:A->Prop)(wellfounded A gtA)->
 ((x:A)((y:A)(gtA x y)->(P y))->(P x))
 ->
 (a:A)(P a).
```

The second induction principle is the principle of well-founded part induction. This induction principle can be applied on the well-founded part of a not neccessarily well-founded set A. This principle works because attention is restricted to the well-founded part of a set A, in which there is no infinite descending sequence. Well-founded part induction is defined as follows:

Definition 4.6 Given a set A and a strict partial order $>$ on A. Suppose it can be proven for all x in W_A that $P(x)$ holds under the assumption that for all y in W_A holds that if $x > y$, then $P(y)$. From this follows that $P(x)$ holds for all x in W_A. Summarized in a scheme:

$$\frac{\forall x \in W_A(\forall y \in W_A : x > y \Rightarrow P(y)) \Rightarrow P(x)}{\forall x \in W_A : P(x)}$$

Note that $\forall y \in W_A$ is redundant here, because from $x \in W_A$ and $x > y$ follows by the definition of well-founded part that $y \in W_A$. Therefore in the formalization of well-founded part induction the clause $\forall y \in W_A$ is replaced by $\forall y \in A$.

In the Coq library well-founded part induction is not available. Remark that well-founded part induction had to be defined for P from `Type` and from `Prop`. For the thesis only P from `Prop` is needed, but in the proof of well-founded part induction the induction with P from `Type` is used. Well-founded part induction is defined with the relation thought of 'greater than' relation, conform the other definitions in this section. Well-founded part induction is formalized in `wf_part_ind` as follows:

```
Theorem wf_part_ind:
 (P:A->Prop)
 ((x:A) (Access A gtA x) -> ((y:A)(gtA x y)->(P y))->(P x))
 ->
 (a:A)(Access A gtA a) -> (P a) .
```

Chapter 5

Lexicographic ordering on a product

5.1 Motivation

The lexicographic ordering is needed in the proof of RPO being well-founded. To motivate why a lexicographic ordering is needed, start with a simple example. Suppose two not neccessarily different sets A and B are given, which are ordered by the strict partial orders $>_A$ and $>_B$ respectively. From the sets A and B pairs (a, b) with $a \in A$ and $b \in B$ can be formed. Usually these elements from the product $A \times B$ are ordered. One way to order these elements is with the lexicographic order. The lexicographic order is based on first comparing the elements from A and when neccessary comparing the elements from B. From this simple case the general case is deduced. Suppose n not neccessarily different sets $S_1, \ldots, S_n$ are given, which are ordered by the strict partial orders $>_{S_1}, \ldots, >_{S_n}$. From these sets elements of the product $S_1 \times \ldots \times S_n$ can be formed. An element of this product is of the form $(s_1, \ldots, s_n)$, with $s_1 \in S_1, \cdots, s_n \in S_n$. These elements are ordered with the n-fold lexicographic order. This order is based on first comparing the elements of S_1, when neccessary comparing the elements of S_2 and so on, until, when neccessary, comparing the elements of S_n.

5.2 Definition of the lexicographic ordering

The lexicographic order of a product of n sets $S_1, \ldots, S_n$ ordered with strict partial orders $>_{S_1}$ $, \ldots, >_{S_n}$ is first defined, because this is the most general lexicographic order. The definition is as follows:

Definition 5.1 Let $S_1, \ldots, S_n$ be sets ordered with strict partial orders $>_{S_1}, \ldots, >_{S_n}$.
The lexicographic order $>_{lex}$ for $s_1, s_1' \in S_1, s_2, s_2' \in S_2 \ldots s_n, s_n' \in S_n$ is given by:
$(s_1, \ldots, s_n) >_{lex} (s_1', \ldots, s_n')$ if and only if $\exists k \leq n(\forall i < k : s_i =_{S_i} s_i') \wedge s_k >_{S_k} s_k'$.

Formalizing this definition in Coq is difficult. The problem is how the define the lexicographic order on the product of n sets, not knowing what n is. Because we need in the proof of well-foundedness of RPO only the lexicographic order on a product of three sets, we formalized the lexicographic order only for the cases $n = 2$ and $n = 3$. When the number of sets in the product is known, the lexicographic order can be defined analoguously to the lexicographic order on the product of three sets. The lexicographic order on a product two sets is also formalized, because the proofs are smaller and the Coq code is easier to understand. The lexicographic order on the product of three sets is denoted as the 3-fold lexicographic order and the lexicographic order on the product of two sets is denoted as the 2-fold lexicographic order. They are defined as follows:

Definition 5.2 Suppose two not neccessarily different sets A and B with strict orders $>_A$ and $>_B$ are given. The 2-fold lexicographic order $>_{lex}$ is given by: $(a, b) >_{lex} (a', b')$ if and only if $(a >_A a') \vee (a =_A a' \wedge b >_B b')$ with $a, a' \in A$ and $b, b' \in B$.

Definition 5.3 Suppose three not neccessarily different sets A, B and C with strict orders $>_A$, $>_B$ and $>_C$ are given. The 3-fold lexicographic order $>_{lex}$ is given by: $(a, b, c) >_{lex} (a', b', c')$ if and only if $(a >_A a') \vee (a =_A a' \wedge b >_B b') \vee (a =_A a' \wedge b =_B b' \wedge c >_C c')$ with $a, a' \in A$, $b, b' \in B$ and $c, c' \in C$.

The 2-fold lexicographic order is already made in Coq. It is found in the standard library in module `Coq.Relations.Relation_Operators`. This definition of the lexicographic order in Coq is based on a dependent product: the second component depends on the first one. The idea behind this is that for the lexicographic order of $A \times B$ first pick an element a of A and dependent on this a pick an element b of B. In this way the pairs from $A \times B$ have the dependency.

This dependent product caused some problems. The main problem was that from $(a, b) = (a, b')$, we cannot conclude $b = b'$. This conclusion is needed for instance when proving irreflexivity of the lexicographic order. The problem is caused by the definition of the dependent product `sigS` for $A \times B$ and the projections, which are called `projS1` and `projS2`. These are found in the module `Coq.Init.Specif`. The definition of `sigS` is as follows:

```
Inductive sigS [A:Set;P:A->Set] : Set
      := existS : (x:A)(P x) -> (sigS A P).
```

This definition states that given a set A and a function P from A to a set (think of this set as the set B), an element a from A and the result from P applied on a form a pair. The second component of the pair is a family $\{P(a)|a : A\}$. In the definition of `projS1` and `projS2` is seen that the second projection forces the second component to be dependent of the first component. This is in the phrase `<[x:(sigS A P)](P (projS1 x))>`.

```
Definition projS1
           := [x:(sigS A P)]Cases x of (existS a _) => a end.
Definition projS2
           := [x:(sigS A P)]<[x:(sigS A P)](P (projS1 x))>
                  Cases x of (existS _ h) => h end.
```

The following Coq code states why from $(a, b) = (a, b')$, we cannot conclude $b = b'$:

```
Lemma equality_sigS :(A:Set; P:(A -> Set); x:A; e,e':(P x))
                     (existS A P x e)=(existS A P x e') -> e=e'.
Proof.
Intros.
Change (projS2 A P (existS A P x e))=(projS2 A P (existS A P x e')).
Rewrite H.
Abort.
```

The tactic `Rewrite H.` should have solved the problem, but it gives the error
`Cannot solve a second order unification problem.`

To overcome the dependency problem, we use a definition of pairs without a dependency between the two components. The idea of this definition is that given two sets A and B, an element from A and an element from B form a pair.The Coq code of this formalization is found in Appendix E. The new formalization of pairs from $A \times B$ is as follows:

```
Inductive sigSpair [A:Set;B:Set] : Set :=
  existSpair : (x:A)(y:B)(sigSpair A B).
```

Using this definition $b = b'$ can be concluded from $(a, b) = (a, b')$. First the projections on the first and second component are defined as follows:

```
Definition projS1pair :=
  [x:(sigSpair A B)]Cases x of (existSpair a _) => a end.

Definition projS2pair :=
  [x:(sigSpair A B)]Cases x of (existSpair _ b) => b end.
```

Example 5.4 A pair from $\mathbb{N} \times \mathbb{N}$ is for instance $(0, 0)$ or $(1, 0)$. For the pair $(1, 0)$ projection on the first component gives 1 and on the second gives 0. See section `Example_pairs` of Appendix E for the Coq code corresponding to this example.

Now everything which is needed from pairs is known, the lexicographic order will be defined. `lexordpair` is similair to lexprod in the module `Coq.Relations.Relation_Operators` except for replacement of the old `sigS` by the new `sigSpair`. In this definition the A and B stand for the sets and the gtA and gtB stand for the relations $>_A$ and $>_B$ on A and B.

```
Inductive lexordpair : (sigSpair A B) -> (sigSpair A B) -> Prop :=
  left_lex_pair :  (a,a':A)(b,b': B)
                   (gtA a a') ->
                   (lexordpair (existSpair A B a b) (existSpair A B a' b'))
| right_lex_pair : (a:A)(b,b':B)
                   (gtB b b') ->
                   (lexordpair (existSpair A B a b) (existSpair A B a b')).
```

Example 5.5 Suppose the strict total order $>$ on natural numbers $\mathbb{N}$ is given. Define the lexicographic product of $\mathbb{N} \times \mathbb{N}$ as follows: $(x, y) >_{lex} (x', y')$ if and only if $(x > x') \vee (x = x' \wedge y > y')$ with $x, x', y, y' \in \mathbb{N}$. In this lexicographic order we have $(1, 0) >_{lex} (0, 1)$ and $(0, 1) >_{lex} (0, 0)$. But $(0, 1) >_{lex} (1, 0)$ and $(0, 0) >_{lex} (0, 0)$ don't hold. See section `Example_lexord_pairs` of Appendix E for the Coq code corresponding to this example.

For the 3-fold lexicographic order first the tripels of the form $A \times B \times C$ are defined. Tripels are already defined in Coq under the name `sigS2`. They are found in the module `Coq.Init.Specif`. But because of the problem caused by the dependency between components, as in the case of pairs, we give a new definition without dependencies. The Coq code corresponding to the formalization of the 3-fold lexicographic order is found in Appendix F. The definitions of `sigStripel` and its projections `projS1tripel`, `projS2tripel` and `projS3tripel` are as follows:

```
Inductive sigStripel[A,B,C:Set]: Set :=
  existStripel : (x:A)(y:B)(z:C)(sigStripel A B C).

Definition projS1tripel :=
  [x:(sigStripel A B C)]Cases x of (existStripel a _ _ ) => a end.

Definition projS2tripel :=
  [x:(sigStripel A B C)]Cases x of (existStripel _ b _ ) => b end.

Definition projS3tripel :=
  [x:(sigStripel A B C)]Cases x of (existStripel _ _ c ) => c end.
```

Example 5.6 A tripel from $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is for instance $(0, 0, 0)$ or $(2, 1, 0)$. For the tripel $(2, 1, 0)$ projection on the first component gives 2, on the second gives 1 and on the third gives 0. See section `Example_tripels` of Appendix F for the Coq code corresponding to this example.

Now the tripels are defined in Coq, the definition of the 3-fold lexicographic order `lexordtripel` is as follows:

```
Inductive lexordtripel : (sigStripel A B C) -> (sigStripel A B C) -> Prop :=

 left_lex_tripel :   (a,a':A)(b,b': B)(c,c':C)
                      (gtA a a') ->
                      (lexordtripel (existStripel A B C a b c)
                                    (existStripel A B C a' b' c'))
| middle_lex_tripel : (a:A)(b,b': B)(c,c':C)
              (gtB b b') ->
                      (lexordtripel (existStripel A B C a b c)
                                    (existStripel A B C a b' c'))
| right_lex_tripel :  (a:A)(b:B)(c,c':C)
                      (gtC c c') ->
                      (lexordtripel (existStripel A B C a b c)
                                    (existStripel A B C a b c')).
```

Example 5.7 Suppose the strict total order $>$ on natural numbers $\mathbb{N}$ is given. Define the 3-fold lexicographic product of three times the natural numbers as follows:$(x, y, z) >_{lex} (x', y', z')$ if and only if $(x > x') \vee (x = x' \wedge y > y') \vee (x = x' \wedge y = y' \wedge z > z')$ with $x, x', y, y', z, z' \in \mathbb{N}$. In this 3-fold lexicographic order we have $(1, 1, 1) >_{lex} (0, 1, 1)$, $(1, 1, 1) >_{lex} (1, 0, 1)$ and $(1, 1, 1) >_{lex} (1, 1, 0)$. But $(0, 0, 0) >_{lex} (0, 0, 0)$ and $(0, 1, 1) >_{lex} (1, 0, 0)$ don't hold. See section `Example_lexord_tripels` of Appendix F for the Coq code corresponding to this example.

5.3 Lexicographic ordering is a strict order

For the general case that the lexicographic ordering is n-fold there is a theorem about strict partial orders. This theorem states that strict partial orders are preserved under making the lexicographic order.

Theorem 5.8 Let $S_1, \ldots, S_n$ be n sets ordered with strict partial orders $>_{S_1}, \ldots, >_{S_n}$. The lexicographic order $>_{lex}$ of these sets is again a strict partial order.

The proof of this theorem is found in Appendix A, because the n-fold lexicographic order isn't formalized in Coq. The theorems for the 2 and 3-fold case are stated below. The Coq code follows both proofs exactly. It's found in Appendices E and F. Both proofs are instances of the proof of the n-fold case.

Theorem 5.9 The 2-fold lexicographic order, with both sets ordered by a strict partial order is again a strict partial order.

Proof
Suppose the two sets A and B are ordered with strict partial orders $>_A$ and $>_B$. We need to prove that the lexicographic order $>_{lex}$ is a strict partial order too. We start with proving that the lexicographic order is irreflexive.
irreflexive
Let (a, b) be an element of $A \times B$. We have to show that $(a, b) >_{lex} (a, b)$ is impossible. Assume $(a, b) >_{lex} (a, b)$ is given. By the definition of the lexiographic product there are two possibilities: $a >_A a$ or $a =_A a \wedge b >_B b$. $a >_A a$ is impossible because $>_A$ is irreflexive and $b >_B b$ is impossible because $>_B$ is irreflexive. So it's impossible to have $(a, b) >_{lex} (a, b)$ and we conclude that $>_{lex}$ is irreflexive.
transitive
Let $(a_1, b_1) >_{lex} (a_2, b_2)$ and $(a_2, b_2) >_{lex} (a_3, b_3)$ be given. We need to prove that we have $(a_1, b_1) >_{lex} (a_3, b_3)$. From $(a_1, b_1) >_{lex} (a_2, b_2)$ follows that either $a_1 >_A a_2$ or $(a_1 =_A a_2$ and $b_1 >_B b_2)$. From $(a_2, b_2) >_{lex} (a_3, b_3)$ follows that either $a_2 >_A a_3$ or $(a_2 =_A a_3$ and $b_2 >_B b_3)$. Combining this gives 4 possibilities.

$a_1 >_A a_2$ and $a_2 >_A a_3$ implies that $a_1 >_A a_3$ by the transitivity of $>_A$. From $a_1 >_A a_3$ follows that $(a_1, b_1) >_{lex} (a_3, b_3)$.

$a_1 >_A a_2$ and ($a_2 =_A a_3$ and $b_2 >_B b_3$) implies that $a_1 >_A a_3$ by replacing a_2 with a_3. From $a_1 >_A a_3$ follows that $(a_1, b_1) >_{lex} (a_3, b_3)$.

($a_1 =_A a_2$ and $b_1 >_B b_2$) and $a_2 >_A a_3$ implies that $a_1 >_A a_3$ by replacing a_2 by a_1. From $a_1 >_A a_3$ follows that $(a_1, b_1) >_{lex} (a_3, b_3)$.

($a_1 =_A a_2$ and $b_1 >_B b_2$) and ($a_2 =_A a_3$ and $b_2 >_B b_3$) implies that ($a_1 =_A a_3$ and $b_1 >_B b_3$) by replacing a_2 by a_3 and by the transitivity of $>_B$. From ($a_1 =_A a_3$ and $b_1 >_B b_3$) follows that $(a_1, b_1) >_{lex} (a_3, b_3)$.

Theorem 5.10 The 3-fold lexicographic order, all three sets ordered by a strict partial order is again a strict partial order.

Proof
Suppose the three sets A, B and C are ordered with strict partial orders $>_A$, $>_B$ and $>_C$. We need to prove that the lexicographic order $>_{lex}$ is a strict partial order too. We start with proving that the lexicographic order is irreflexive.
irreflexive
Let (a, b, c) be an element of $A \times B \times C$. We have to show that $(a, b, c) >_{lex} (a, b, c)$ is impossible. Assume $(a, b, c) >_{lex} (a, b, c)$ is given. By the definition of the lexiographic order there are three possibilities: $a >_A a$ or $a =_A a \wedge b >_B b$ or $a =_A a \wedge b =_B b \wedge c >_C c$. $a >_A a$ is impossible because $>_A$ is irreflexive, $b >_B b$ is impossible because $>_B$ is irreflexive and $c >_C c$ is impossible because $>_C$ is irreflexive. So it's impossible to have $(a, b, c) >_{lex} (a, b, c)$ and we conclude that $>_{lex}$ is irreflexive.
transitive
Let $(a_1, b_1, c_1) >_{lex} (a_2, b_2, c_2)$ and $(a_2, b_2, c_2) >_{lex} (a_3, b_3, c_3)$ be given. We have to prove that holds $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$. From $(a_1, b_1, c_1) >_{lex} (a_2, b_2, c_2)$ follows that either $a_1 >_A a_2$ or ($a_1 =_A a_2$ and $b_1 >_B b_2$) or ($a_1 =_A a_2$ and $b_1 =_B b_2$ and $c_1 >_C c_2$). From $(a_2, b_2, c_2) >_{lex}$ (a_3, b_3, c_3) follows that either $a_2 >_A a_3$ or ($a_2 =_A a_3$ and $b_2 >_B b_3$) or ($a_2 =_A a_3$ and $b_2 =_B b_3$ and $c_2 >_C c_3$). Combining these gives 9 possibilities.

$a_1 >_A a_2$ and $a_2 >_A a_3$ implies that $a_1 >_A a_3$ by the transitivity of $>_A$. From $a_1 >_A a_3$ follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

$a_1 >_A a_2$ and ($a_2 =_A a_3$ and $b_2 >_B b_3$) implies that $a_1 >_A a_3$ by replacing a_2 with a_3. From $a_1 >_A a_3$ follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

$a_1 >_A a_2$ and ($a_2 =_A a_3$ and $b_2 =_B b_3$ and $c_2 >_C c_3$) implies that $a_1 >_A a_3$ by replacing a_2 with a_3. From $a_1 >_A a_3$ follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

($a_1 =_A a_2$ and $b_1 >_B b_2$) and $a_2 >_A a_3$ implies that $a_1 >_A a_3$ by replacing a_2 by a_1. From $a_1 >_A a_3$ follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

($a_1 =_A a_2$ and $b_1 >_B b_2$) and ($a_2 =_A a_3$ and $b_2 >_B b_3$) implies that ($a_1 =_A a_3$ and $b_1 >_B b_3$) by replacing a_2 by a_3 and by the transitivity of $>_B$. From ($a_1 =_A a_3$ and $b_1 >_B b_3$) follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

($a_1 =_A a_2$ and $b_1 >_B b_2$) and ($a_2 =_A a_3$ and $b_2 =_B b_3$ and $c_2 >_C c_3$) implies that
($a_1 =_A a_3$ and $b_1 >_B b_3$) by replacing a_2 by a_3 and b_2 by b_3. From ($a_1 =_A a_3$ and $b_1 >_B b_3$) follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

($a_1 =_A a_2$ and $b_1 =_B b_2$ and $c_1 >_C c_2$) and $a_2 >_A a_3$ implies $a_1 >_A a_3$ by replacing a_2 with a_1. From $a_1 >_A a_3$ follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

$(a_1 =_A a_2$ and $b_1 =_B b_2$ and $c_1 >_C c_2)$ and $(a_2 =_A a_3$ and $b_2 >_B b_3)$ implies that $(a_1 =_A a_3$ and $b_1 >_B b_3)$ by replacing a_2 by a_3 and b_2 by b_1. From $(a_1 =_A a_3$ and $b_1 >_B b_3)$ follows that $(a_1, b_1, c_1) >_{lex} (a_3, b_3, c_3)$.

$(a_1 =_A a_2$ and $b_1 =_B b_2$ and $c_1 >_C c_2)$ and $(a_2 =_A a_3$ and $b_2 =_B b_3$ and $c_2 >_C c_3)$ implies $(a_1 =_A a_3$ and $b_1 =_B b_3$ and $c_1 >_C c_3)$ by replacing a_2 with a_3, b_2 with b_3 and by the transitivity of $>_C$.

5.4 Well-foundedness of the lexicographic ordering

In the next theorem the well-foundedness of the n-fold lexicographic order is covered.

Theorem 5.11 A lexicographic order of n sets which are all ordered by a well-founded strict partial order is again well-founded.

The proof of this theorem is found in Appendix A, because the n-fold lexicographic order isn't formalized in Coq. For the 2 and 3-fold case the theorem is proved below. The proofs are just instances of the proof of the n-fold case. In Coq both these proofs are formalized; the formal proofs are the 'upside-down' version of the proofs on paper. This is done because presenting the proof on paper this way is intuitively better to understand. The formalization is found in Appendices E and F.

Theorem 5.12 A 2-fold lexicographic order, with sets which are both ordered by a well-founded strict partial order is again well-founded.

Proof
Suppose the two sets A and B are ordered with well-founded strict partial orders $>_A$ and $>_B$. To prove that the lexicographic order $>_{lex}$ is well-founded too, is the same as proving $\forall a \in A \;\; \forall b \in B : (a, b) \in W_{A \times B}$. This is stated in the following lemma:

Lemma 5.13 $\forall a \in A \;\; \forall b \in B : (a, b) \in W_{A \times B}$.

Proof
The proof proceeds by well-founded induction on a. Remark that this is possible, because A is well-founded. Lemma 5.13 is just the conclusion of applying well-founded induction with $P(a) = \forall b \in B : (a, b) \in W_{A \times B}$. If $\forall a \in A(\forall a' \in A : a > a' \Rightarrow P(a')) \Rightarrow P(a)$ holds, well-founded induction can be applied and this lemma is proven. This is stated in the following lemma:

Lemma 5.14 $\forall a \in A(\forall a' \in A : a > a' \Rightarrow P(a')) \Rightarrow P(a)$.

Proof
First assume an arbitrary $a_0 \in A$ and assume $\forall a' \in A : a_0 > a' \Rightarrow P(a')$. To show that $P(a_0)$ holds. Remember that $P(a_0) = \forall b \in B : (a_0, b) \in W_{A \times B}$. The rest of the proof proceeds by well-founded induction on b with $Q(b) = (a_0, b) \in W_{A \times B}$. If $\forall b \in B(\forall b' \in B : b > b' \Rightarrow Q(b')) \Rightarrow Q(b)$ holds, well-founded induction can be applied and this lemma is proven. This is stated in the following lemma:

Lemma 5.15 $\forall b \in B(\forall b' \in B : b > b' \Rightarrow Q(b')) \Rightarrow Q(b)$.

Proof
First assume an arbitrary $b_0 \in B$ and assume $\forall b' \in B : b_0 > b' \Rightarrow Q(b')$. To show that $Q(b_0)$ holds. Remember that $Q(b_0) = (a_0, b_0) \in W_{A \times B}$. Using the definition of the well-founded part, (a_0, b_0) is in $W_{A \times B}$ if it can be proven that for all (c, d) with $(a_0, b_0) >_{lex} (c, d)$ it holds that (c, d) is in $W_{A \times B}$. $(a_0, b_0) >_{lex} (c, d)$ comes from two cases:

$a_0 > c$.
In the proof of Lemma 5.14 it is assumed that $\forall a' \in A : a_0 > a' \Rightarrow P(a')$ holds. The c here is such an a' and thus conclude that $P(c) = \forall b \in B : (c, b) \in W_{A\times B}$ holds. Hence it follows that $(c, d) \in W_{A\times B}$ holds.

$a_0 = c \wedge b_0 > d$.
In the proof of Lemma 5.15 it is assumed that $\forall b' \in B : b_0 > b' \Rightarrow Q(b')$ holds. The d here is such a b' and thus conclude that $Q(d) = (a_0, d) \in W_{A\times B}$ holds. Because $a_0 = c$, it follows that $(c, d) \in W_{A\times B}$.

Summarizing this proof gives that because Lemma 5.15 is proven, applying well-founded induction two times proves Lemma 5.13, which is equivalent to Theorem 5.12.

Theorem 5.16 A 3-fold lexicographic order, with all three sets ordered by a well-founded strict partial order is again well-founded.

Proof
Suppose the three sets A, B and C are ordered with well-founded strict partial orders $>_A$, $>_B$ and $>_C$. We need to prove that the lexicographic product $>_{lex}$ is well-founded too, this is the same as proving $\forall a \in A \;\; \forall b \in B \;\; \forall c \in C : (a, b, c) \in W_{A\times B\times C}$. This is stated in the following lemma:

Lemma 5.17 $\forall a \in A \;\; \forall b \in B \;\; \forall c \in C : (a, b, c) \in W_{A\times B\times C}$.

Proof
The proof proceeds by well-founded induction on a. Remark that this is possible, because A is well-founded. Lemma 5.17 is just the conclusion of applying well-founded induction with
$P(a) = \forall b \in B \;\; \forall c \in C : (a, b, c) \in W_{A\times B\times C}$. If $\forall a \in A(\forall a' \in A : a > a' \Rightarrow P(a')) \Rightarrow P(a)$ holds, well-founded induction can be applied and this lemma is proven. This is stated in the following lemma:

Lemma 5.18 $\forall a \in A(\forall a' \in A : a > a' \Rightarrow P(a')) \Rightarrow P(a)$.

Proof
First assume an arbitrary $a_0 \in A$ and assume $\forall a' \in A : a_0 > a' \Rightarrow P(a')$. To show that $P(a_0)$ holds. Remember that $P(a_0) = \forall b \in B \;\; \forall c \in C : (a_0, b, c) \in W_{A\times B\times C}$. The rest of this proof proceeds by well-founded induction on b with $Q(b) = \forall c \in C : (a_0, b, c) \in W_{A\times B\times C}$. If $\forall b \in B(\forall b' \in B : b > b' \Rightarrow Q(b')) \Rightarrow Q(b)$ holds, well-founded induction can be applied and this lemma is proven. This is stated in the following lemma:

Lemma 5.19 $\forall b \in B(\forall b' \in B : b > b' \Rightarrow Q(b')) \Rightarrow Q(b)$.

Proof
First assume an arbitrary $b_0 \in B$ and assume $\forall b' \in B : b_0 > b' \Rightarrow Q(b')$. To show that $Q(b_0)$ holds. Remember that $Q(b_0) = \forall c \in C : (a_0, b_0, c) \in W_{A\times B\times C}$. The rest of this proof proceeds by well-founded induction on c with $R(c) = (a_0, b_0, c) \in W_{A\times B\times C}$.
If $\forall c \in C(\forall c' \in C : c > c' \Rightarrow R(c')) \Rightarrow R(c)$ holds, well-founded induction can be applied and this lemma is proven. This is stated in the following lemma:

Lemma 5.20 $\forall c \in C(\forall c' \in C : c > c' \Rightarrow R(c')) \Rightarrow R(c)$.

Proof
First assume an arbitrary $c_0 \in C$ and assume $\forall c' \in C : c > c' \Rightarrow R(c')$. To show that $R(c_0)$ holds. Remember that $R(c_0) = (a_0, b_0, c_0) \in W_{A\times B\times C}$. Using the definition of the well-founded part, (a_0, b_0, c_0) is in $W_{A\times B\times C}$ if it can be proven that for all (k, l, m) with $(a_0, b_0, c_0) >_{lex} (k, l, m)$ it holds that (k, l, m) is in $W_{A\times B\times C}$. $(a_0, b_0, c_0) >_{lex} (k, l, m)$ comes from three cases:

$a_0 > k$.
In the proof of Lemma 5.18 it is assumed that $\forall a' \in A : a_0 > a' \Rightarrow P(a')$ holds. The k here is such an a' and thus conclude that $P(k) = \forall b \in B \;\; \forall c \in C : (k, b, c) \in W_{A\times B\times C}$ holds. Hence it follows that $(k, l, m) \in W_{A\times B\times C}$.

$a_0 = k \wedge b_0 > l$
In the proof of Lemma 5.19 it is assumed that $\forall b' \in B : b_0 > b' \Rightarrow Q(b')$ holds. The l here is such an b' and thus conclude that $Q(l) = \forall c \in C : (a_0, l, c) \in W_{A\times B\times C}$ holds. Because $a_0 = k$, it follows that $(k, l, m) \in W_{A\times B\times C}$.

$a_0 = k \wedge b_0 = l \wedge c_0 > m$
In the proof of Lemma 5.20 it is assumed that $\forall c' \in C : c > c' \Rightarrow R(c')$ holds. The m here is such an c' and thus conclude that $R(m) = (a_0, b_0, m) \in W_{A\times B\times C}$ holds. Because $a_0 = k$ and $b_0 = l$, it follows that $(k, l, m) \in W_{A\times B\times C}$.

Summarizing this proof gives that because Lemma 5.20 is proven, applying well-founded induction three times proves Lemma 5.17, which is equivalent to Theorem 5.16.

Chapter 6

Multisets and multiset ordering

6.1 Definition of multisets

Before defining the multiset order first has to be known what a multiset exactly is. In words and a bit sloppy a multiset is 'a set with repeated elements'. Sometimes multisets are referred to as the abstract data type bag. The formal definition is as follows:

Definition 6.1 A multiset M over a set A is a function $M : A \to \mathbb{N}$. In this definition $M(x)$ is the number of copies of x. $M(x)$ is often called the multiplicity of x.

Remark that a multiset is finite if there is only a finite number of x with $M(x) > 0$. A notation for all finite multisets over a set A is $\mathcal{M}(A)$. Denote the multiset consisting of two a's as $\{\{a, a\}\}$. The empty multiset with $M(a) = 0$ for all $a \in A$ is denoted by $\emptyset$. In the remainder of this thesis attention is restricted to finite multisets only.

Example 6.2 Some examples of finite multisets over $\mathbb{N}$ are:

$$\emptyset$$
$$\{\{2, 1, 1\}\}$$
$$\{\{3, 2, 1\}\}$$
$$\{\{3, 1\}\}$$
$$\{\{6, 3, 3, 3, 2\}\}$$
$$\{\{5, 5, 4, 2, 2, 1, 1, 1, 1, 1, 1\}\}$$

In Coq multisets are already formalized in the library in module `Coq.Sets.Multiset`. At first this formalization was used, but it gave problems. In the definition from the library a multiset could be infinite too. For instance $\mathbb{N}$ is a multiset according to the definition from the library. This was not a major problem, because we could restrict to finite multisets only. But the major problem was about the equality. In the library an equality between the elements of A is assumed. But this equality wasn't preserved under making multisets. In general if $a = a'$ holds, $\{\{a\}\} = \{\{a'\}\}$ doesn't follow. This is undesirable. A new definition of multisets is made in such a way that it restricts to finite multisets only. This resulted in viewing the multiset as a data type list. Lists are already formalized in the library in module `Coq.Lists.PolyList`. A multiset over a set A is then formalized as follows:

```
(list A)
```

It is important to note that equality on multisets is different from equality on lists. Equality on list is often defined as: Two lists p and q are equal if the head of p is equal to the head of q and if the tail of p is equal to the tail of q. Remember that the head of a list over a set A is an element from A and the tail is again a list of elements from A. Note that the order of the elements in the

list is important here.
Equality on multisets is different, because the order of the elements in a multiset doesn't matter. Two multisets M and N over A are equal if for all elements a from A it holds that the multiplicity of a in M is equal to the multiplicity of a in N. Before equality of multisets is formalized in Coq, multiplicity is formalized. The Coq code corresponding to this Chapter is found in Appendix G.

Multiplicity is a counter which compares a to each element of the multiset M. If the element is equal to a the multiplicity is increased by one and otherwise the multiplicity stays equal. The multiplicity is the same as $M : A \to \mathbb{N}$ in Definition 6.1. For this equality of elements from A needs to be decidable. This will be a hypothesis. This hypothesis is needed in all sections of formalization of multisets.

```
Hypothesis Aeq_dec : (x,y:A){eq A x y} + {~(eq A x y)}.
```

In Coq multiplicity is now formalized as:

```
Fixpoint multiplicity [a:A;m:(list A)] : nat :=
Cases m of
  nil => O
| (cons b n) => Cases (Aeq_dec a b) of
                  (left _ )  => (S (multiplicity a n))
                | (right _ ) => (multiplicity a n)
                  end
end.
```

Besides equality, there are more interesting operations and relations on the set of finite multisets $\mathcal{M}(A)$. The most interesting are stated below:

Definition 6.3 Operations and relations on multisets:

equality:	$M = N \Leftrightarrow$ forall $a \in A : M(a) = N(a)$.
size:	$\text{size}(M) = \sum_{\{a \mid M(a)>0\}} M(a)$.
element:	$a \in M \Leftrightarrow M(a) > 0$.
inclusion:	$M \subseteq N \Leftrightarrow \forall a \in A : M(a) \leq N(a)$.
union:	$(M \cup N)(a) = M(a) + N(a)$.
remove:	$(M - a_0)(a) = \begin{cases} M(a) \dot{-} 1 & \text{if } a_0 = a \\ M(a) & \text{if } a_0 \neq a. \end{cases}$
difference:	$(M - N)(a) = M(a) \dot{-} N(a)$
	where $m \dot{-} n = \begin{cases} m - n & \text{if } m \geq n \\ 0 & \text{if } m < n. \end{cases}$

In Coq first equality of multisets is formalized. It is shown in Coq that equality is an equivalence relation. Equality is formalized as follows:

```
Definition eq_M : (list A) -> (list A) -> Prop :=
  [m,n:(list A)] (a:A) (multiplicity a m) = (multiplicity a n) .
```

The size of a multiset isn't needed in the thesis, but to be complete it corresponds to the length of the list which represents the multiset. The length of a list `length` is found in `Coq.Lists.PolyList`. If an element a of A is in the list which represents a multiset, then the element is in the multiset too. Element is formalized by `In` from `Coq.Lists.PolyList`. Inclusion follows from element and is not formalized, because it wasn't needed.

The union of two multisets corresponds to the union of the lists which represent the multisets. Union of lists is already formalized in `Coq.Lists.PolyList` and is denoted with `app`. Removing an element from a multiset is different, because there wasn't an equivalent formalization for removing an element from a list in the library. Removing an element from a multiset certainly does

correspond with removing an element from the list which represents the multiset. Removing an element from a multiset is formalized as follows:

```
Fixpoint remove_M [a:A;m:(list A)]: (list A) :=
Cases m of
  nil => (nil A)
| (cons b n) => Cases (Aeq_dec a b) of
                  (left _ ) => n
                | (right _ ) => (cons b (remove_M a n))
                  end
end .
```

First note that decidability of equality on A is needed for `remove_M`. Remark that `remove_M` only formalizes removing an element from a multiset and not removing an element from an arbitrary list. Because in a list the order of the elements does matter, formalizing remove will be more complicated. Removing the first occurrence of an element a of A from a list of elements from A will result in a different list then when the second occurrence of a was removed. For multisets this will result in the same multiset and that's why `remove_M` could be formalized this way. Difference follows from remove and is not formalized, because it wasn't needed.

The formalization in Coq of this multiset operations and relations contains many additional hypotheses. This is due to the fact the notion of equality on multisets is new. The interactions between this new equality, accessibility, adding elements to multisets, removing elements from multisets, determining if an element is in a multiset and union of multisets causes the hypotheses. The hypotheses can been seen as elementary properties for the multisets that have to hold. Fortunately they are all straightforward and when multiset would be formalized fully, they should all be possible to prove.

6.2 Ordering on multisets

The ordering on multisets is defined as follows:

Definition 6.4 Suppose a strict partial order $>$ on a set S is given. In the multiset order $>_{mul}$ on $\mathcal{M}(S)$, $M >_{mul} N$ holds for $M, N \in \mathcal{M}(S)$ if and only if:
There exist $X, Y \in \mathcal{M}(S)$ such that:

$$\emptyset \neq X \subseteq M$$

$$N = (M - X) \cup Y$$

$$\forall y \in Y \exists x \in X : x > y$$

In words: $M >_{mul} N$, if M is obtained from N by doing the following one or more times: Remove an element x and replace it with a finite number of elements all smaller than x in $>$.

Example 6.5 For the multisets of Example 6.2 the following holds:

$\{\{3,2,1\}\}$	$>_{mul}$	$\{\{2,1,1\}\}$	$X = \{\{3,2\}\}$	$Y = \{\{2,1\}\}$
$\{\{3,2,1\}\}$	$>_{mul}$	$\{\{2,1,1\}\}$	$X = \{\{3\}\}$	$Y = \{\{1\}\}$
$\{\{3,2,1\}\}$	$>_{mul}$	$\{\{3,1\}\}$	$X = \{\{2\}\}$	$Y = \emptyset$
$\{\{6,3,3,3,2\}\}$	$>_{mul}$	$\{\{5,5,4,3,3,2,1,1\}\}$	$X = \{\{6,3\}\}$	$Y = \{\{5,5,4,1,1\}\}$

In the last case of Example 6.5 the 6 is replaced by {{5,5,4}}, 3 is replaced by {{1,1}}. Remark that there are several possibilities for X and Y, as is seen in the first two cases. The following doesn't hold:

$$\{\{3,2,1\}\} \quad >_{mul} \quad \{\{3,2,1\}\} \quad \text{because } X = \emptyset \text{ is impossible}$$

Formalizing Definition 6.4 in Coq presented problems. One problem is that there are many different X and Y from where $M >_{mul} N$ can be concluded. X and Y are not uniquely determined. To make this problem somewhat smaller the notion of multiset reduction is introduced. In words multiset reduction is: Remove an element x and replace it with a finite number of elements all smaller than x in $>$. The definition is as follows:

Definition 6.6 Let $>$ be a strict partial order on a set A. In the multiset reduction $>_{MUL}$ on $\mathcal{M}(A)$, $M >_{MUL} N$ holds for $M, N \in \mathcal{M}(A)$ if and only if:
There exists $M_0 \in \mathcal{M}(A)$ and $a \in A$ such that:

$$M = M_0 \cup \{\{a\}\}$$

$$\exists K \in \mathcal{M}(A) : (N = M_0 \cup K) \wedge (a >_{MUL} K)$$

with $a >_{MUL} K$ defined as $a >_{MUL} K \Leftrightarrow \forall k \in K : a > k$.

Remark that the multiset order is the transitive closure of this multiset reduction. A consequence of using this multiset reduction is that transtivity is lost.

Example 6.7 In general the multiset reduction is not transitive. Both $\{\{1,2,3\}\} >_{MUL} \{\{1,2,2\}\}$ and $\{\{1,2,2\}\} >_{MUL} \{\{0,2,2\}\}$ hold in the multiset reduction, but $\{\{1,2,3\}\} >_{MUL} \{\{0,2,2\}\}$ doesn't hold.

The notion of multiset reduction from Definition 6.6 is formalized in Coq. First recall that we work under the assumption that equality on A is decidable.

```
Hypothesis Aeq_dec : (x,y:A){eq A x y} + {~(eq A x y)} .
```

In Coq, the definition of $>_{MUL}$ on $A \times \mathcal{M}(A)$ is defined as follows:

```
Definition gt_EM : A -> (list A) -> Prop :=
  [a:A][m:(list A)](b:A)(In b m) -> (gtA a b) .
```

Here `gtA` is a greater than relation on A. Now multiset reduction is defined inductively as follows:

```
Inductive gt_M : (list A) -> (list A) -> Prop :=
  gtm : (a:A)(n,m,m',k: (list A))
          (gt_EM a k) ->
          (eq_M A Aeq_dec m (cons a m')) ->
          (eq_M A Aeq_dec n (app k m')) ->
          (gt_M m n)
```

If we have $M >_{MUL} N$ then $M = M_0 \cup \{\{a\}\}$ and $N = M_0 \cup K$ with $a >_{MUL} K$. So M is not empty. Thus M is always of the form $M = M' \cup \{\{a'\}\}$ for an $a' \in A$ and a $M' \in mathcal M(A)$. This gives two possibilities for the multiset reduction, namely that $a = a'$ and $a \neq a'$. If $a = a'$ then the reduction replaces exactly the element a' from M. If $a \neq a'$ the reduction replaces another element from M and leaves a' unchanged. In Coq this case distinction is made by elimination of the hypotheisis `Aeq_dec`, which translates in proving `gt_M` for $a' = a$ and $a' \neq a$.

In the Coq code there are two hypotheses generated by `gt_M`. These hypotheses state the interaction between `eq_M` and `gt_M` and between `remove_M`, `In` and `gt_M`.

Strict partial orders are preserved under the multiset order. Only the proof on paper is stated because the multiset order isn't formalized in Coq.

Theorem 6.8 The multiset order of a strict partial order is again a strict partial order.

Proof
Let $>$ be a strict partial order, to show that the multiset order is a strict partial order. First irreflexivity is shown and then transitivity is concluded.
Irreflexive
Assume $M >_{MUL} M$. To prove that this leads to a contradiction. There is a $M_0 \in \mathcal{M}(A)$ and an $a \in A$ with $M = M_0 \cup \{a\}$ and $\exists K \in \mathcal{M}(A) : (M = M_0 \cup K) \wedge (a > K)$. So $M = M_0 \cup \{a\}$ and $M = M_0 \cup K$ holds. This is only possible when $K = \{a\}$. But then $a > a$ has to hold and that is impossible, because $>$ is irreflexive.
Transitive
Because the multiset order is the transitive closure of the multiset reduction it's trivially transitive. For an explicit proof see [BN98].

The multiset reduction is irreflexive. This is not proved in Coq, because the file `Multisets.v` contains still too many hypotheses. The multiset reduction is not transitive. Hence the multiset reduction is not a strict partial order.

6.3 Well-foundedness of the multiset ordering

This section is concerned with well-foundedness of the multiset order and the multiset reduction. First we show well-foundedness of the multiset reduction, following closely the proof due to Buchholz presented by Nipkow in [Nip98]. The main difference is that our presentation of the proof is the other way around, starting with the desired main result that multiset reduction is well-founded. This is because we find it better to understand in that way. The proof of well-foundedness of the multiset reduction in formalized in Coq, following the presentation in [Nip98].

Theorem 6.9 The multiset reduction $>_{MUL}$ of a well-founded strict partial order $>$ on A is well-founded.

The theorem states that if all $a \in A$ are in W_A, then all multisets in $\mathcal{M}(A)$ are in $W_{\mathcal{M}(A)}$ with respect to the multiset reduction. We write W instead of $W_{\mathcal{M}(A)}$. The Theorem 6.9 gets the following form:

Theorem 6.10

$$\text{If } \forall a \in A : a \in W_A \text{ then } \forall M \in \mathcal{M}(A) : M \in W.$$

Proof
First let M be an arbitrary multiset from $\mathcal{M}(A)$. It's enough to prove that: If all elements of M are in W_A then M is in W. This is stated in the following lemma:

Lemma 6.11

$$\text{If } \forall m \in M : m \in W_A \text{ then } M \in W.$$

Proof
The proof proceeds by induction on the size of M.

> **base case:** $\text{size}(M) = 0$, that means that $M = \emptyset$. To prove that $\emptyset \in W$. But this is neccessarily true, because there are no N with $N <_{MUL} \emptyset$. So for all N with $N <_{MUL} \emptyset$ holds that $N \in W$ and hence by the definition of the well-founded part we have that $\emptyset \in W$.
>
> **induction step:** $\text{size}(M) > 0$. Assume $\forall m \in M : m \in W_A \Rightarrow M \in W$. we have to prove $\forall m \in M \cup \{\{a\}\} : m \in W_A \Rightarrow M \cup \{\{a\}\} \in W$. Denote $\forall a \in A : a \in W_A$ with $\forall a \in W_A$ and denote $\forall M \in \mathcal{M}(A) : M \in W$ with $\forall M \in W$. This is equivalent with proving $\forall a \in W_A \;\; \forall M \in W : M \cup \{\{a\}\} \in W$. This is possible because the hypothesis $\forall m \in M : m \in W_A \Rightarrow M \in W$ is always true due to the clause $\forall M \in W$. The hypothesis $\forall m \in M \cup \{\{a\}\, m \in W_A$ is also always true due to the clause $\forall a \in W_A$. Summarizing: if Lemma 6.12 can be proven, then this lemma is proven.

Lemma 6.12

$$\forall a \in W_A \;\; \forall M \in W : M \cup \{\{a\}\} \in W.$$

Proof
The proof proceeds by well-founded part induction on a. We have to show the conclusion of applying well-founded part induction with $P(a) = \forall M \in W : M \cup \{\{a\}\} \in W$. If $\forall a \in W_A(\forall a' \in W_A : a > a' \Rightarrow P(a')) \Rightarrow P(a)$ holds, well-founded part induction can be applied and this lemma is proven. This is stated in the following lemma:

Lemma 6.13

$$\begin{array}{l}\forall a \in W_A[(\forall a' \in W_A : a > a' \Rightarrow \\ (\forall M \in W : M \cup \{\{a'\}\} \in W)) \Rightarrow \\ \forall M \in W : M \cup \{\{a\}\} \in W].\end{array}$$

Proof
This proof proceeds by well-founded part induction on M. Remark that here only well-founded part induction is possible, because it isn't certain yet that all elements M from $\mathcal{M}(A)$ are well-founded. We have to show the conclusion of applying well-founded part induction on M with $P(M) = M \cup \{\{a\}\} \in W$. If $\forall M \in W(\forall M' \in W : M > M' \Rightarrow P(M')) \Rightarrow P(M)$ holds, well-founded part induction can be applied and then this lemma is proven. This is stated in Lemma 6.14. In Lemma 6.14 the hypothesis $(\forall a' \in W_A : a > a' \Rightarrow (\forall M \in W : M \cup \{\{a'\}\} \in W))$ is still present. The rest of the lemma contains the upper part of well-founded part induction on M.

Lemma 6.14

$$\begin{array}{c}\text{If} \\ \forall a \in W_A[(\forall a' \in W_A : a > a' \Rightarrow (\forall M' \in W : M' \cup \{\{a'\}\} \in W)) \\ \text{then if} \\ \forall M \in W(\forall M >_{MUL} M' : M' \cup \{\{a\}\} \in W)] \\ \text{then} \\ M \cup \{\{a\}\} \in W\end{array}$$

Proof
This proof proceeds by the definition of the well-founded part of a set. Assume $a \in W_A$, $(\forall a' \in W_A : a > a' \Rightarrow (\forall M' \in W : M' \cup \{\{a'\}\} \in W))$, $M \in W$ and $\forall M \in W(\forall M >_{MUL} M' : M' \cup \{\{a\}\} \in W)$. To prove that $M \cup \{\{a\}\}$ is in the well-founded part of $\mathcal{M}(A)$.
If $\forall N : M \cup \{\{a\}\} >_{MUL} N \Rightarrow N \in W$ can be proven, the lemma is proven. Two cases are distinguished:

The first case is that N is of the form $M' \cup \{\{a\}\}$, with $M >_{MUL} M'$. This handles the case that a isn't replaced in the multiset reduction $M \cup \{\{a\}\} >_{MUL} N$. From the assumption $(\forall M >_{MUL} M' : M' \cup \{\{a\}\} \in W)$ follows that $N \in W$.

The second case is that N is of the form $M \cup K$ with $K < \{\{a\}\}$. This handles the case that a is replaced in the multiset reduction $M \cup \{\{a\}\} >_{mul} N$. $N \in W$ is proven by induction on the size of K.

Base case Let size$(K) = 0$, then $K = \emptyset$. Then follows that $N = M$ and $M \in W$ by the third assumption of this proof. Thus also $N \in W$.

Induction step Let size$(K) > 0$, then K is of the form $K' \cup \{\{a'\}\}$, with K' a multiset and $a' \in W_A$ with $a > a'$. To prove from the hypothesis $M \cup K' \in W$ that $M \cup K' \cup \{\{a'\}\} \in W$. Use the assumption $(\forall a' \in W_A : a > a' \Rightarrow (\forall M' \in W : M' \cup \{\{a'\}\} \in W))$ to conclude that $M \cup K' \cup \{\{a'\}\} \in W$. From this follows $N \in W$.

Because 6.14 is proven, a complete proof of 6.11 is given by applying well-founded part induction two times and applying induction on the structure of terms.

In Coq Theorem 6.9 is formalized in `multisets_wellfounded`. This is the last theorem in the Coq file `Multisets.v`. The proof uses Lemma 6.11, which is formalized in `elements_inW_implies_multiset_inW`. The annotation in both proofs should sufficy to understand the Coq code. Lemma 6.12 and Lemma 6.13 are both proven with well-founded part induction and formalized in `lemma_one` and `lemma_two`. Finally Lemma 6.14 is formalized in `lemma_three`. In the proof of `lemma_three` a lot hypotheses are used. The proof of `lemma_three` is complex and not transparent, due to the use of so much hypotheses.

The remaining question is whether well-foundedness is preserved for the multiset order. Well-foundedness is only proved for the multiset reduction. If well-foundedness is preserved under taking the transitive closure, it can be concluded that it's preserved under taking the multiset order. Well-foundedness is indeed preserved under taking the transitive closure. This is because the transitive closure leaves the lengths of the sequences equal or makes them smaller. So when alle descending sequences are finite, all descending sequences in the transitive closure are also finite. This shows that the multiset order is well-founded too.

Chapter 7

Recursive Path Order

7.1 RPO on the set of terms

This chapter is concerned with the recursive path order (RPO). RPO is an ordering on first-order terms, that is induced by an ordering on the signature. In this chapter RPO is defined, it's shown that RPO is a strict partial order, a reduction order and that RPO is well-founded. The definition of RPO and the well-foundedness of RPO are formalized in Coq. The Coq code corresponding to this chapter is found in Appendix H.

In the literature different definitions of a precedence are found. Some authors define a precedence as a strict partial order on a signature. Others define a precedence as a well-founded strict partial order on a signature. In this thesis the following definition is chosen:

Definition 7.1 A precedence is a strict partial order on a signature.

In the definition of terms in Coq, the signature is chosen to be `nat`. The consequence of this choice is that the precendence on the signature is chosen to be the usual relation $>$ in the natural numbers. In Coq this relation is denoted by `gt`. Note that the relation $>$ is a total order, whereas the precedence is not required to be a total order. So we consider a slightly more restricted case in the formalizations. RPO is formalized for only the signature of the natural numbers with the total order $>$.

RPO is defined starting from a well-founded precedence $\succ$ on the set of function symbols Σ. RPO is defined as follows:

Definition 7.2 Suppose a well-founded precedence $\succ$ on the signature Σ is given. Then we have for $s, t \in \mathcal{T}(\Sigma, X)$ that $s >_{rpo} t$ holds if one of the following three clauses is true:

1. $s = f(s_1, \ldots, s_m)$ for some $m \geq 0$
   $t = g(t_1, \ldots, t_n)$ for some $n \geq 0$
   $f \succ g$
   $s >_{rpo} t_i$ for all $1 \leq i \leq n$

2. $s = f(s_1, \ldots, s_m)$ for some $m \geq 0$
   $t = f(t_1, \ldots, t_n)$ for some $n \geq 0$
   $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{t_1, \ldots, t_n\}\}$
   $s >_{rpo} t_i$ for all $1 \leq i \leq n$

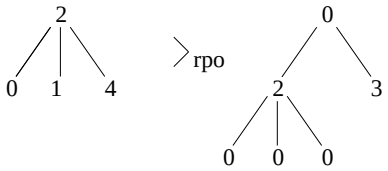
3. $s = f(s_1, \ldots, s_m)$ for some $m > 0$
   $s_i \geq_{rpo} t$ for some $1 \leq i \leq m$

A lot can be said about this definition. First the notation $>_{rpo,mul}$ denotes the multiset order of $>_{rpo}$. The recursion is seen in the call for $>_{rpo}$ in each of the three cases. $\geq_{rpo}$ is the reflexive closure of $>_{rpo}$. In other words: $s \geq_{rpo} t \Leftrightarrow (s >_{rpo} t$ or $s = t)$. Remark that if all function symbols have fixed arity, then clause 2 can only be applied if the arity of the function symbols is greater than or equal to 1. This is because $\emptyset >_{rpo,mul} \emptyset$ doesn't hold.
Note that clause 1 of RPO can only be applied to terms $f(s_1, \ldots, s_m)$ and $g(t_1, \ldots, t_n)$ with $f \neq g$, because the precedence is well-founded and a strict partial order.
RPO satisfies the subterm property. This is proven in the following lemma, which is also formalized in Coq:

Lemma 7.3 For all $s_i \in s_1, \ldots, s_n$ we have $f(s_1, \ldots, s_n) >_{rpo} s_i$.

Proof
Let s_i be from $s_1, \ldots, s_n$. Because $s_i = s_i$ holds, case 3 of RPO can be applied.

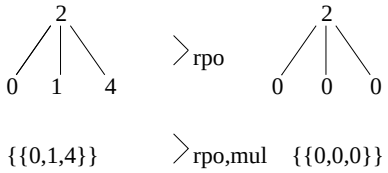
Example 7.4 Given the two terms $s = 2(0, 1, 4)$ and $t = 0(2(0, 0, 0), 3)$. We show that $s >_{rpo} t$ holds.



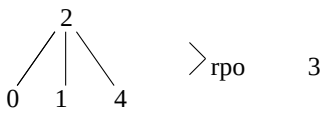
This in done in three steps. The term trees for step two and three are added, to show the structure of the terms. Hopefully this example is better understood with help of these trees.

1. The first step consists of using $2 \succ 0$. Applying case 1 of the definition of RPO, we need to show that $s >_{rpo} t_i$ for all $1 \leq i \leq n$. For this example this means that we need to show $2(0, 1, 4) >_{rpo} 2(0, 0, 0)$ and $2(0, 1, 4) >_{rpo} 3$. These cases are considered in item 2 and 3.

2. For $2(0, 1, 4) >_{rpo} 2(0, 0, 0)$ we are in RPO case 2. We need to show that $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{t_1, \ldots, t_n\}\}$. For this example this means that we need to show $\{\{0, 1, 4\}\} >_{rpo,mul} \{\{0, 0, 0\}\}$. $1 >_{rpo} 0$ and $4 >_{rpo} 0$ are true according to RPO case 1. And thus from the definition of the multiset ordering we conclude $\{\{0, 1, 4\}\} >_{rpo,mul} \{\{0, 0, 0\}\}$. There also has to hold that $s >_{rpo} t_i$ for all $1 \leq i \leq n$. This means for this example that we need to show $2(0, 1, 4) >_{rpo} 0$ three times. This is true according to RPO case 1.



3. For $2(0, 1, 4) >_{rpo} 3$ we are in RPO case 3, because case 1 and 2 can't apply here. For RPO case 3 we need to show $s_i \geq_{rpo} t$ for some $1 \leq i \leq m$. Take for this s_i the number 4. According to RPO case 1 we have $4 >_{rpo} 3$.



Summarizing this example gives that $2(0, 1, 4) >_{rpo} 0(2(0, 0, 0), 3)$ for RPO which uses the multiset ordering.

Example 7.5 In this example some boundary cases of RPO are considered. Consider a signature $\Sigma = \{c, d, g, f\}$ with precedence $c \succ d$, $c \succ g$, $f \succ c$, $f \succ d$ and $f \succ g$.

First of all: $x \not>_{rpo} s$, where x is variable and s an arbitrary term. This is because in all clauses of $>_{rpo}$, the left term has to contain a function symbol and x doesn't contain a function symbol.

The next six examples to consider are the ones where the left term is a constant symbol. Note that the third case of $>_{rpo}$ cannot apply to these examples, because in this case the left term has to have at least one subterm. In the next table five examples are explained:

question	**answer**	**explanation**
$c >_{rpo} x$?	no	x is not of the form $t = g(t_1, \ldots, t_n)$, so cases 1 and 2 don't apply
$c >_{rpo} c$?	no	Only case 2 can be applied, but $\emptyset >_{rpo,mul} \emptyset$ doesn't hold: case 2 fails.
$c >_{rpo} d$?	yes	Case 1 can be applied, using $c \succ d$.
$c >_{rpo} g(x)$?	no	Using $c \succ g$ case 1 can be applied, but $c >_{rpo} x$ doesn't hold: case 1 fails.
$c >_{rpo} g(c)$?	no	Using $c \succ g$ case 1 can be applied, but $c >_{rpo} c$ doesn't hold: case 1 fails.

Remains the question whether $c >_{rpo} g(t_1, \ldots, t_n)$ holds. The answer is in most cases no. This is because case 1 can only applied several times and c doesn't change. After these steps one of the cases $c >_{rpo} c$?, $c >_{rpo} x$? and $c >_{rpo} d$? is reached. If case $c >_{rpo} d$? is reached, the answer is yes and in all other cases the answer is no.

Conclusion: In RPO a constant is only greater than a term of the form a constant which is smaller in $\succ$ or a term which consists only of constants which are smaller in $\succ$.
The next examples are the ones with on the left side only $f(x)$.

question	**answer**	**explanation**
$f(x) >_{rpo} x$?	yes	Apply case 3: $x \geq_{rpo} x$.
$f(x) >_{rpo} c$?	yes	Using $f \succ c$, apply case 1.
$f(x) >_{rpo} f(x)$?	no	Apply case 2, $\{\{x\}\} >_{rpo,mul} \{\{x\}\}$ doesn't hold: case 2 fails.
$f(x) >_{rpo} f(c)$?	no	Apply case 2, $\{\{x\}\} >_{rpo,mul} \{\{c\}\}$ doesn't hold: case 2 fails.
$f(x) >_{rpo} g(x)$?	yes	Using $f \succ g$, apply case 1: $f(x) >_{rpo} x$ holds.
$f(x) >_{rpo} g(c)$?	yes	Using $f \succ g$, apply case 1: $f(x) >_{rpo} c$ holds.

The last examples are the ones with left side only $f(c)$.

question	**answer**	**explanation**
$f(c) >_{rpo} x$?	no	Only case 3 can be applied: $c \geq_{rpo} x$ doesn't hold: case 3 fails.
$f(c) >_{rpo} c$?	yes	Using $f \succ c$, apply case 1.
$f(c) >_{rpo} f(x)$?	no	Apply case 2, $\{\{c\}\} >_{rpo,mul} \{\{x\}\}$ doesn't hold: case 2 fails.
$f(c) >_{rpo} f(c)$?	no	Apply case 2, $\{\{c\}\} >_{rpo,mul} \{\{c\}\}$ doesn't hold: case 2 fails.
$f(c) >_{rpo} g(x)$?	no	Using $f \succ g$, apply case 1: $f(c) >_{rpo} x$ doesn't hold.
$f(c) >_{rpo} g(c)$?	yes	Using $f \succ g$, apply case 1: $f(c) >_{rpo} c$ holds.

Unfortunately all above boundary cases couldn't be formalized in Coq, because the file `Multisets.v` is still full with hypotheses, which makes calculating with multisets impossible.

7.2 Formalizing RPO in Coq

In Coq RPO is formalized using the multiset reduction in case 2, instead of the multiset ordering. This is because in the file `Multiset.v` only the multiset reduction is formalized. Formalizing RPO in Coq needs decidability of equality on ground terms. This is because the set of proper subterms is seen as a finite multiset. And when determining whether two multisets are equal, decidability of equality is needed. In Coq we work with the following hypothesis, because we are going to use parts of `Multisets.v`, where decidability of equality is needed.

```
Hypothesis termeq_dec : (s,t:term){s=t}+{~s=t} .
```

In Coq RPO on ground terms has been formalized inductively as follows:

```
Inductive rpo : term -> term  -> Prop :=
rpo_one :
  (f,g:nat)(ss,ts:termlist)
  (gt f g)
  ->
  ((ti:term) (in_termlist ti ts) -> (rpo (buildterm  f ss) ti))
  ->
  (rpo (buildterm  f ss) (buildterm  g ts))
|
rpo_two :
  (f:nat)(si:term)(ss,ts,us:termlist)
  (in_termlist si ss)
  ->
  ((ui:term)(in_termlist ui us) -> (rpo si ui))
  ->
  (eq_M term termeq_dec
    (termlist_to_list ts)
    (app (termlist_to_list us)
         (rem term termeq_dec si (termlist_to_list ss))
    )
  )
  ->
  ((ti:term) (in_termlist ti ts) -> (rpo (buildterm  f ss) ti))
  ->
  (rpo (buildterm f ss) (buildterm f ts))
|
rpo_three_a :
  (f:nat)(si,t:term )(ss:termlist)
  (rpo si t)
  ->
  (in_termlist  si ss)
  ->
  (rpo (buildterm f ss) t)
|
rpo_three_b :
  (f:nat)(si,t:term)(ss:termlist)
  (eq term si t)
  ->
  (in_termlist si ss)
  ->
  (rpo (buildterm f ss) t).
```

Some remarks to this formalization are to be made. The first clause follows the definition of RPO exactly. From $f \succ g$ and $f(s_1, \ldots, s_m) >_{rpo} t_i$ for all $1 \leq i \leq n$ follows that $f(s_1, \ldots, s_m) >_{rpo} g(t_1 \ldots, t_n)$.

The second clause is somewhat more complicated. The problem with this clause is that the multiset reduction requires RPO as argument. Just simply using `gt_M` which is defined in the Coq file `Multisets.v`, will make the inductive definition recursive, which is not accepted by Coq. The order from which the multiset reduction is made, is the RPO order on terms, but this order is still not completely defined at this moment. This problem is solved by writing the comparision of two multisets out in the definition of RPO. This definition is accepted by Coq. What happens

now is that multiset comparison in RPO is written out in three conditions instead of using `gt_M`. Multiset reduction (see also Definition 6.6) in RPO is formalized with the hypotheses:

1. `(in_termlist si ss)`
2. `((ui:term)(in_termlist ui us) -> (rpo si ui))`
3. `(eq_M term termeq_dec (termlist_to_list ts)`
   `(app (termlist_to_list us)`
   `(rem term termeq_dec si (termlist_to_list ss))`

Item 1 corresponds to the condition `(eq_M A Aeq_dec m (cons a m'))` from `gt_M`. Item 2 corresponds to `(gt_EM a k)` and item 3 corresponds to `(eq_M A Aeq_dec n (app k m'))`.

Concerning the third clause the expression `(rpo si t) \/ (eq term si t)` which is natural for formalizing $s_i \geq_{rpo} t$, gave problems in Coq too. In this case the third clause of RPO is formalized as:

```
rpo_three:
  (f:nat)(si,t:term )(ss:termlist)
  (rpo si t)\/ (eq term si t)
  ->
  (in_termlist  si ss)
  ->
  (rpo  (buildterm f ss) t).
```

The problem is that sometimes clause 3 of RPO is recursively called for (when using `(rpo si t)`) and sometimes not (when using `(eq term si t)`). The reaction of Coq on this is to ignore the recursive call, which makes the formalization different from the definition. This is seen when `rpo_ind` is checked, the case concerning the third clause is then as follows:

```
Check rpo_ind.

->((f:nat; si,t:term; ss:termlist)
            (rpo si t)\/si=t
            ->(in_termlist si ss)
            ->(P (buildterm f ss) t))
```

Coq has ignored the recursive call and this is seen in the fact that the property P doesn't have to hold for s_i and t. Defining the third case in two parts solves the problem. The third case is then formalized as:

```
rpo_three_a :
  (f:nat)(si,t:term )(ss:termlist)
  (rpo si t)
  ->
  (in_termlist  si ss)
  ->
  (rpo  (buildterm f ss) t)
|
rpo_three_b :
  (f:nat)(si,t:term)(ss:termlist)
  (eq term si t)
  ->
  (in_termlist si ss)
  ->
  (rpo (buildterm f ss) t).
```

Checking whether induction is well defined, is done using `Check` in Coq. The case concerning the third clause is then as follows:

```
->((f:nat; si,t:term; ss:termlist)
            (rpo si t)
            ->(P si t)
            ->(in_termlist si ss)
            ->(P (buildterm f ss) t))
->((f:nat; si,t:term; ss:termlist)
            si=t->(in_termlist si ss)->(P (buildterm f ss) t))
```

This case states that P indeed has to hold for s_i and t.

Remark that RPO which is formalized in Coq isn't a strict partial order. Because the multiset reduction was formalized, transitivity is lost. See Example 6.7. The RPO which is formalized is actually a refinement of the original RPO. Comparison which was done by case 2 in one step in the original RPO, is now done by comparisons in several steps of case 2 in the formalized RPO. Strict formally spoken the formalized RPO has to be called recursive path reduction. But for proving well-foundedness transitivity isn't needed, so the names aren't changed.

Example 7.6 Considering Example 7.4 again, we conclude that using the multiset reduction $2(0,1,4) >_{rpo} 0(2(0,0,0),3)$ doesn't hold. This is because $\{\{0,1,4\}\} >_{rpo,MUL} \{\{0,0,0\}\}$ doesn't hold, because two elements are replaced in one step. For the multiset reduction this is impossible.

Example 7.7 In the formalized RPO holds $2(1,1) >_{rpo} 2(0,1)$ and $2(0,1) >_{rpo} 2(0,0)$. But $2(1,1) >_{rpo} 2(0,0)$ doesn't hold because two elements are replaced. In the definition of RPO $2(1,1) >_{rpo} 2(0,0)$ does hold.

Example 7.8 In the formalized RPO holds $1(0,0) >_{rpo} 1(0)$ and $1(0) >_{rpo} 1()$. But $1(0,0) >_{rpo} 1()$ doesn't hold because two elements are deleted. In the definition of RPO $1(0,0) >_{rpo} 1()$ does hold.

Examples 7.7 and 7.8 show that the formalized RPO isn't transitive.

7.3 RPO is a strict partial order

The proof of RPO being a strict partial order is not formalized. The proof is due to Dershowitz, see [Der82]. Note that Theorem 7.9 is about RPO which uses the multiset ordering. In Coq Theorem 7.9 isn't formalized, because when formalizing irreflexivity some calculation with multisets is needed, which isn't complete due to the hypotheses in `Multisets.v`. Transitivity can't be formalized because the formalized RPO uses multiset reduction, which isn't transitive.

Theorem 7.9 When the precedence $\succ$ is a strict partial order, then RPO is a strict partial order too.

Proof
First irreflexivity is shown and then transitivity.
irreflexive
Suppose a term $s \in \mathcal{T}(\Sigma, X)$ is given with $s >_{rpo} s$. To prove that this leads to a contradiction. We proceed by induction on the structure of terms.

> $s \in X$.
> When s is a variable, conclude that $s >_{rpo} s$ will be never true, because for all cases of RPO at least the left-hand side had to be a term with a function symbol in it. See also Example 7.5.

$s = f(s_1, \ldots, s_n)$.
The induction hypothesis here is that $>_{rpo}$ is irreflexive for all $s_1, \ldots, s_n$. Because irreflexivity is preserved under making the multiset out of $s_1, \ldots, s_n$, the induction hypothesis translates to $>_{rpo,mul}$ is irreflexive. To prove that $f(s_1, \ldots, s_n) >_{rpo} f(s_1, \ldots, s_n)$ leads to a contradiction. Already known is that $\{\{s_1, \ldots, s_n\}\} >_{rpo,mul} \{\{s_1, \ldots, s_n\}\}$ is impossible and from this follows that $f(s_1, \ldots, s_n) >_{rpo} f(s_1, \ldots, s_n)$ is impossible.

transitive
Suppose that $s >_{rpo} t$ and $t >_{rpo} u$. We need to show that $s >_{rpo} u$. The proof proceeds by well-founded induction on $|s| + |t| + |u|$. There are three possibilities in which $s >_{rpo} t$ can have been obtained, and three possibilities in which $t >_{rpo} u$ can have been obtained. This yields nine possibilities for the combination. Three cases collaps to a single one, so we need to consider seven cases.

1. $s >_{rpo} t$ with RPO case 1 and $t >_{rpo} u$ with RPO case 1.

   We have $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$ with $f \succ g$ and $s >_{rpo} t_i$ for all $1 \leq i \leq n$. Further, $u = h(u_1, \ldots, u_p)$ with $g \succ h$ and $t >_{rpo} u_i$ for all $1 \leq i \leq p$.

   Because $\succ$ is transitive, we have $f \succ h$. If $p = 0$, we conclude $s >_{rpo} u$ by RPO case 1. If $p > 0$, let $i \in \{1, \ldots, p\}$. We have $s >_{rpo} t$ and $t >_{rpo} u_i$. Because $|s| + |t| + |u_i| < |s| + |t| + |u|$follows from the induction hypothesis that $s >_{rpo} u_i$. Hence we have $s > u$ by RPO case 1.

2. $s >_{rpo} t$ with RPO case 1 and $t >_{rpo} u$ with RPO case 2.

   We have $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$ with $f \succ g$ and $s >_{rpo} t_i$ for all $1 \leq i \leq n$. Further, $u = g(u_1, \ldots, u_p)$ with $\{\{t_1, \ldots, t_n\}\} >_{rpo,mul} \{\{u_1, \ldots, u_p\}\}$ and $t >_{rpo} u_i$ for all $1 \leq i \leq p$.

   If $p = 0$, we conclude $s >_{rpo} u$ by RPO case 1. If $p > 0$, let $i \in \{1, \ldots, p\}$. We have $s >_{rpo} t$ and $t >_{rpo} u_i$. Because $|s| + |t| + |u_i| < |s| + |t| + |u|$ follows from the induction hypothesis that $s >_{rpo} u_i$. Hence we have $s > u$ by RPO case 1.

3. $s >_{rpo} t$ with RPO case 1 and $t >_{rpo} u$ with RPO case 3.

   We have $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$ with $f \succ g$ and $s >_{rpo} t_i$ for all $1 \leq i \leq n$. Further, we have $t_j \geq_{rpo} u$ for some $j \in \{1, \ldots, n\}$.

   From $s >_{rpo} t_j$ and $t_j \geq_{rpo} u$ and the induction hypothesis we conclude that $s >_{rpo} u$.

4. $s >_{rpo} t$ with RPO case 2 and $t >_{rpo} u$ with RPO case 1.

   We have $s = f(s_1, \ldots, s_m)$ and $t = f(t_1, \ldots, t_n)$ with $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{t_1, \ldots, t_n\}\}$ and $s >_{rpo} t_i$ for all $1 \leq i \leq n$. Further, we have $u = h(u_1, \ldots, u_p)$ with $g \succ h$ and $t >_{rpo} u_i$ for all $1 \leq i \leq p$.

   If $p = 0$ we conclude $s >_{rpo} u$ by RPO case 1. If $p > 0$, let $i \in \{1, \ldots, p\}$. We have $s >_{rpo} t$ and $t >_{rpo} u_i$. Because $|s| + |t| + |u_i| < |s| + |t| + |u|$ follows from the induction hypothesis that $s >_{rpo} u_i$. Hence we have $s > u$ by RPO case 1.

5. $s >_{rpo} t$ with RPO case 2 and $t >_{rpo} u$ with RPO case 2.

   This is the most complicated case. We have $s = f(s_1, \ldots, s_m)$ and $t = f(t_1, \ldots, t_n)$ with $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{t_1, \ldots, t_n\}\}$ and $s >_{rpo} t_i$ for all $1 \leq i \leq n$. Further, $u = f(u_1, \ldots, u_p)$ with $\{\{t_1, \ldots, t_n\}\} >_{rpo,mul} \{\{u_1, \ldots, u_p\}\}$ and $t >_{rpo} u_i$ for all $1 \leq i \leq p$.

We need to show $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{u_1, \ldots, u_p\}\}$ in order to conclude $s >_{rpo} u$ with RPO case 2. This is done by reasoning as in the proof of transitivity of the multiset ordering, see [BN98].

6. $s >_{rpo} t$ with RPO case 2 and $t >_{rpo} u$ with RPO case 3.
   We have $s = f(s_1, \ldots, s_m)$ and $t = f(t_1, \ldots, t_n)$ with $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{t_1, \ldots, t_n\}\}$ and $s >_{rpo} t_i$ for all $1 \leq i \leq n$. Further, $t_j \geq_{rpo} u$ for some $j \in \{1, \ldots, n\}$.

   We have $s >_{rpo} t_j$ and $t_j \geq u$. Because $|s| + |t_j| + |u| < |s| + |t| + |u|$ follows from the induction hypothesis that $s >_{rpo} u$.

7. $s >_{rpo} t$ with RPO case 3.
   We have $s = f(s_1, \ldots, s_m)$ with $s_j \geq_{rpo} t$ for some $j \in \{1, \ldots, m\}$.(So $m > 0$.) Here it doesn't matter how $t >_{rpo} u$ is obtained. From $s \geq_{rpo} t$ and $t >_{rpo} u$ and the induction hypothesis we conclude $s >_{rpo} u$.

7.4 RPO is a reduction order

For RPO being a reduction order has to be proven that RPO is closed under substitutions and that RPO is closed under contexts. The following two theorems show that RPO is a reduction order. Note that both Theorem 7.10 and 7.11 are about RPO with the multiset ordering. Theorem 7.10 and 7.11 aren't formalized in Coq, because the notions of substitution and contexts aren't formalized either.

Theorem 7.10 RPO is closed under substitutions: for every $s, t \in \mathcal{T}(\Sigma, X)$ holds if $s >_{rpo} t$ then $\sigma(s) >_{rpo} \sigma(t)$ for all substitutions σ.

Proof
We proceed by induction on $|s| + |t|$. Consider the three cases of RPO from which $s >_{rpo} t$ could come. Let σ be an arbitrary substitution.

1. $s = f(s_1, \ldots, s_m)$ for some $m \geq 0$
   $t = g(t_1 \ldots, t_n)$ for some $n \geq 0$
   $f \succ g$
   $s >_{rpo} t_i$ for all $1 \leq i \leq n$

   Because $|s| + |t_i| < |s| + |t|$ the induction hypothesis can be applied on $s >_{rpo} t_i$ and thus follows $\sigma(s) >_{rpo} \sigma(t_i)$. Applying case 1 of RPO gives $\sigma(s) >_{rpo} \sigma(t)$.

2. $s = f(s_1, \ldots, s_m)$ for some $m \geq 0$
   $t = f(t_1, \ldots, t_n)$ for some $n \geq 0$
   $\{\{s_1, \ldots, s_m\}\} >_{rpo,mul} \{\{t_1, \ldots, t_n\}\}$
   $s >_{rpo} t_i$ for all $1 \leq i \leq n$

   For every t_j either there is a s_i with $s_i >_{rpo} t_j$ or there is an s_i with $s_i = t_j$. In the first case we have by the induction hypothesis that $\sigma(s_i) \geq_{rpo} \sigma(t_j)$. We conclude $\sigma(\{\{s_1, \ldots, s_m\}\}) >_{rpo} \sigma(\{\{t_1, \ldots, t_n\}\})$. Because $|s| + |t_i| < |s| + |t|$ the induction hypothesis can be applied on $s >_{rpo} t_i$ and thus follows $\sigma(s) >_{rpo} \sigma(t_i)$. Applying case 2 of RPO gives $\sigma(s) >_{rpo} \sigma(t)$.

3. $s = f(s_1, \ldots, s_m)$ for some $m > 0$.
   $s_i \geq_{rpo} t$ for some $1 \leq i \leq m$

   Because $|s_i| + |t| < |s| + |t|$ the induction hypothesis can be applied on $s_i \geq_{rpo} t$. And thus follows $\sigma(s_i) \geq_{rpo} \sigma(t)$. Applying case 3 of RPO gives $\sigma(s) >_{rpo} \sigma(t)$.

Theorem 7.11 RPO is closed under contexts: for every $s, t \in \mathcal{T}(\Sigma, X)$ holds if $s >_{rpo} t$ then $f(\ldots s \ldots) >_{rpo} f(\ldots t \ldots)$ for all contexts $f(\ldots \Box \ldots)$.

Proof
Suppose $f(\ldots \Box \ldots)$ is an arbitrary context. Putting s in this context gives $f(\ldots s \ldots)$. Putting t in this context gives $f(\ldots t \ldots)$. Note that $\{\{\ldots s \ldots\}\} >_{rpo,mul} \{\{\ldots t \ldots\}\}$ can only hold when $s >_{rpo} t$ holds, because the only difference between the multisets lies in s and t. And indeed $s >_{rpo} t$ holds, thus is concluded $\{\{\ldots s \ldots\}\} >_{rpo,mul} \{\{\ldots t \ldots\}\}$. Note that also the subterm property holds, thus $f(\ldots s \ldots) >_{rpo} s$. Using the transitivity of RPO and $s >_{rpo} t$ gives $f(\ldots s \ldots) >_{rpo} t$. And the subterm property holds for all other subterms not equal to s too: $f(\ldots s_k \ldots) >_{rpo} s_k$ for $s_k \neq s$. Because all these subterms also occur in $f(\ldots t \ldots)$ and are just all subterms not equal to t follows that $f(\ldots s \ldots) >_{rpo} t_i$ for all t_i which are subterms of $f(\ldots t \ldots)$. Applying case 2 of RPO with $\{\{\ldots s \ldots\}\} >_{rpo,mul} \{\{\ldots t \ldots\}\}$ and $f(\ldots s \ldots) >_{rpo} t_i$ gives that holds $f(\ldots s \ldots) >_{rpo} f(\ldots t \ldots)$.

7.5 Well-foundedness of RPO on the set of terms

In this section we show that RPO is well-founded. Because we have shown well-foundedness of multiset reduction only, we consider RPO with multiset reduction instead of RPO with multiset ordering. The proof method is taken from [JR99]: We show that every term is accessible for RPO by induction on the definition of terms. The proof doesn't use Kruskal's tree theorem.

The proof is organized in the same way as the proof of the multiset reduction being well-founded. As a consequence the formalization in Coq is again the other way around.

Theorem 7.12 RPO on the set of terms $\mathcal{T}(\Sigma, X)$ is well-founded.

For this theorem it needs to be proven that every element of $\mathcal{T}(\Sigma, X)$ is accessible, i.e. is in the well-founded part W of $\mathcal{T}(\Sigma, X)$ ordered by RPO. This is stated in Theorem 7.13:

Theorem 7.13

$$\forall s \in \mathcal{T}(\Sigma, X) : s \in W$$

Proof
The proof proceeds by induction on the structure of terms.

> **base case:** s is a variable. Because there are no elements from $\mathcal{T}(\Sigma, X)$ which are smaller than s, the hypothesis $\forall s' \in \mathcal{T}(\Sigma, X) s > s' : s' \in W$ is true. We conclude that $s \in W$.
>
> **induction step:** $s = f(s_1, \ldots, s_n)$. From the induction hypothesis that all $s_1, \ldots, s_n$ are in W has to follow that $f(s_1, \ldots, s_n)$ is in W. This is stated in the following lemma:

Lemma 7.14 If $s_1, \ldots, s_m$ are in W and $s = f(s_1, \ldots, s_m)$ for some $m \geq 0$, then s is in W.

Proof
The lemma follows from Lemma 7.15 and the definition of the well-founded part.

Lemma 7.15 If $s_1, \ldots, s_m$ are in W and $s = f(s_1, \ldots, s_m) >_{rpo} t$ for some $m \geq 0$, then t is in W.

Proof
The proof proceeds by well-founded induction on tripels of the form $(f, \{\{s_1, \ldots, s_m\}\}, t)$, where all $s_1, \ldots, s_m$ have to be in W. Here f is a function symbol from Σ ordered with $\succ$. $\{\{s_1, \ldots, s_m\}\}$ are multisets of terms from $\mathcal{T}(\Sigma, X)$ ordered with the multiset reduction based on RPO. Finally t is a term from $\mathcal{T}(\Sigma, X)$ ordered by the length of the term, which is ordered by the usual order $>$ on $\mathbb{N}$. Tripels of this sort are ordered lexicographicly by $>_{lex}$. The relation between the arguments

of the tripel is that for every tripel $(f, \{\{s_1, \ldots, s_m\}\}, t)$ has to hold $f(s_1, \ldots, s_m) >_{rpo} t$.

Note that Σ and $\mathcal{T}(\Sigma, X)$ are well-founded with respect to $\succ$ and $>$ (order using the length of a term). Provided that $s_1, \ldots, s_m$ are in W, the multiset built from $s_1, \ldots, s_m$ is in $W_{\mathcal{M}(\mathcal{T}(\Sigma,X))}$. (Using Theorem 6.10 with A equal to the set s_i's from $\mathcal{T}(\Sigma, X)$ which are in W). From this follows that the set of all multisets built from terms that are in W is well-founded with respect to the multiset reduction. From all elements of the tripel being well-founded follows that $>_{lex}$ is well-founded too. (Using Theorem 5.16) Hence it's legal to use the well-founded induction.
The proof actually starts here.
Suppose $s = f(s_1, \ldots, s_m)$ for some $m \geq 0$ and $s_1, \ldots, s_m$ are in W. Assume $s >_{rpo} t$. To prove: t is in W. As already said, the proof proceeds by wellfounded induction on $(f, \{\{s_1, \ldots, s_m\}\}, t)$. The induction hypothesis is thereby: for all tripels $(f', \{\{s'_1, \ldots, s'_n\}\}, t')$ with
$(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (f', \{\{s'_1, \ldots, s'_n\}\}, t')$ and indeed $s'_1, \ldots, s'_n$ are in W holds: t' is in W.
From this hypothesis is proven for $(f, \{\{s_1, \ldots, s_m\}\}, t)$, that t is in W. Following the definition of $>_{rpo}$, three cases are distinguished:

Case 1 of RPO
$t = g(t_1, \ldots, t_n)$ for some $n \geq 0$ and $f \succ g$ and $f(s_1, \ldots, s_m) >_{rpo} t_i$ for all $1 \leq i \leq n$

The idea is to go one step lower into the tripels. Suppose there is an arbitrary u with $g(t_1, \ldots, t_n) >_{rpo} u$. Using the induction hypothesis with
$(f', \{\{s'_1, \ldots, s'_n\}\}, t') = (g, \{\{t_1, \ldots, t_n\}\}, u)$ leaves to prove three things:

1. $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (g, \{\{t_1, \ldots, t_n\}\}, u)$.
2. $t_1, \ldots, t_n$ are in W.
3. $g(t_1, \ldots, t_n) >_{rpo} u$, the connection between the arguments of a tripel.

When all three above cases are proved, conclude from the induction hypothesis that u is in W. And because u was arbitrary with $g(t_1, \ldots, t_n) >_{rpo} u$, it follows from the definition of the well-founded part that $g(t_1, \ldots, t_n)$ is in W, and hence t is in W. The proofs of the three cases are given by:

1. To prove: $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (g, \{\{t_1, \ldots, t_n\}\}, u)$. This indeed holds by the definition of the lexicographic product and the hypothesis $f \succ g$.
2. To prove: $t_1, \ldots, t_n$ are in W. For this apply Lemma 7.16 which is stated after the proof of well-foundedness of RPO.
3. To prove: $g(t_1, \ldots, t_n) >_{rpo} u$. This is true, because it's an hypothesis.

Case 2 of RPO
$t = f(t_1, \ldots, t_n)$ and $\{\{s_1, \ldots, s_m\}\} >_{rpo,MUL} \{\{t_1, \ldots, t_n\}\}$ and $f(s_1, \ldots, s_m) >_{rpo} t_i$ for all $1 \leq i \leq n$

This proof follows the same pattern as the proof of case 1 of RPO. Suppose an arbitrary u with $f(t_1, \ldots, t_n) >_{rpo} u$ is given. Note that the arity is not fixed, which explains the use of m and n in $f(s_1, \ldots, s_m)$ and $f(t_1, \ldots, t_n)$. Using the induction hypothesis with $(f', \{\{s'_1, \ldots, s'_n\}\}, t') = (f, \{\{t_1, \ldots, t_n\}\}, u)$ leaves to prove three things:

1. $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (f, \{\{t_1, \ldots, t_n\}\}, u)$.
2. $t_1, \ldots, t_n$ are in W
3. $f(t_1, \ldots, t_n) >_{rpo} u$, the connection between the arguments of a tripel.

When all three above cases are proved, conclude from the induction hypothesis that u is in W. And because u was arbitrary with $f(t_1, \ldots, t_n) >_{rpo} u$, it follows from the definition of the well-founded part that $f(t_1, \ldots, t_n)$ is in W, and hence t is in W. The proofs of the three cases are given by:

1. To prove: $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (f, \{\{t_1, \ldots, t_n\}\}, u)$. This indeed holds by the definition of the lexicographic product and the hypothesis $\{\{s_1, \ldots, s_m\}\} >_{rpo,MUL} \{\{t_1, \ldots, t_n\}\}$.
2. To prove: $t_1, \ldots, t_n$ are in W. For this apply Lemma 7.16 which is stated after the proof of well-foundedness of RPO.
3. To prove: $f(t_1, \ldots, t_n) >_{rpo} u$. This is true, because it's an hypothesis.

Case 3 of RPO
$s_i \geq_{rpo} t$ for some $1 \leq i \leq m$

Because all s_i are in W, and there is a s_i with $s_i \geq_{rpo} t$, it follows from the Remark 4.2 that t has to be in W.

The proof of well-foundedness of RPO isn't complete yet, because two times Lemma 7.16 is applied, which is stated below:

Lemma 7.16
Suppose:

1. a tripel $(f, \{\{s_1, \ldots, s_m\}\}, t)$ with all $s_1, \ldots, s_m$ in W and $t = g(t_1, \ldots t_n)$.

2. for all tripels $(f', \{\{s'_1, \ldots, s'_n\}\}, t')$ with $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (f', \{\{s'_1, \ldots, s'_n\}\}, t')$ and indeed $s'_1, \ldots, s'_n$ are in W holds: t' is in W.
(The induction hypothesis of the well-founded part induction).

3. $f(s_1, \ldots, s_m) >_{rpo} t_i$ for all t_i's from $t_1, \ldots, t_n$.
(From the case distinction of RPO)

Check whether when this lemma is applied in the proof of well-foundedness of RPO that indeed all these hypotheses are satisfied. From these hypotheses follows that $t_1, \ldots, t_n$ are in W.

Proof
Let t_i be arbitrary from $t_1, \ldots, t_n$. Apply the induction hypothesis with $(f', \{\{s'_1, \ldots, s'_n\}\}, t') = (f, \{\{s_1, \ldots, s_m\}\}, t_i)$. This induction hypothesis leaves to prove three things:

1. $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (f, \{\{s_1, \ldots, s_m\}\}, t_i)$.
2. $s_1, \ldots, s_m$ are in W
3. $f(s_1, \ldots, s_m) >_{rpo} t_i$, the connection between the arguments of a tripel.

When all three above cases are proved, conclude from the induction hypothesis that t_i is in W. And because t_i was arbitrary follows that all $t_1, \ldots, t_n$ are in W. The proof of the three cases are given by:

1. To prove: $(f, \{\{s_1, \ldots, s_m\}\}, t) >_{LEX} (f, \{\{s_1, \ldots, s_m\}\}, t_i)$. This indeed holds by the definition of the lexicographic product and because $|t| > |t_i|$ for all subterms t_i of t.
2. To prove: $s_1, \ldots, s_m$ are in W. This is true, because it's an hypothesis.
3. To prove: $f(s_1, \ldots, s_m) >_{rpo} t_i$. This is true, because it's an hypothesis.

In Coq this proof is formalized in file `RPO.v`. Note that the formalization of the proof is exactly the other way around as it's presented here. The first theorem here is the last theorem in Coq. Before discussing the formalization of the proof, first is explained how the list of well-founded terms is formalized and why well-founded induction in Lemma 7.15 can indeed be applied. This

is found in the beginning of section `rpo_wellfounded` of the Coq file `RPO.v`. The tripels which are used in Lemma 7.15 are of the form (function symbol, multiset of well-founded terms, term). Because multisets are formalized as lists the form of the tripels in the formalization is (function symbol, list of well-founded terms, term). Well-founded induction is only allowed when the tripels are well-founded. According to Theorem 5.11 the induction is allowed when the elements of the tripel are well-founded.

The first component of the tripel is the set `nat`. That `nat` is well-founded is proven in the library in module `Coq.Arith.Wf_nat`. In the library the ordering is oriented as 'less than' and in this thesis as 'greater than'. That `nat` is well-founded is added as an hypothesis called `wfnat`.

The second component is the set of termlists consisting of well-founded (for RPO) terms, ordered with the multiset reduction of RPO. For this first is defined when a term is in W and when a termlist is in W. In Coq this is formalized as:

```
Definition term_wf [s:term] : Prop :=
  (Access term (rpo termeq_dec) s) .

Definition termlist_wf [ss:termlist] : Prop :=
  (si:term)(in_termlist si ss)->(Access term (rpo termeq_dec) si) .
```

A termlist which satisfies `termlist_wf` is defined using `sig` from the library from module `Coq.Init.Specif`. The idea of `sig` is that it connects a set with a property on this set. `sig` represents the subset of a set in which all elements fullfill a property P. In this case the set is equal to `termlist` and the property equal to `termlist_wf`. In Coq this gives:

```
Definition WTL : Set :=
  (sig termlist termlist_wf) .
```

The first projection `proj1_sig` of a `WTL` will give the termlist, which fullfills `termlist_wf`. On the set `WTL` an order `gt_WTL` using the order `gt_M` on multisets is formalized as follows:

```
Definition gt_WTL : WTL -> WTL -> Prop :=
  [ss,ts:WTL]
  (gt_M term (rpo termeq_dec) termeq_dec
    (termlist_to_list (proj1_sig termlist termlist_wf ss))
    (termlist_to_list (proj1_sig termlist termlist_wf ts))
  ) .
```

After defining this, it has to be shown that `WTL` is wellfounded with respect to `gt_WTL`. This is shown in lemma `wfWTL`. This lemma uses two other lemma's and a hypothesis in Coq. The hypothesis `acces_for_lists` states that if all elements from a list are accessible, then the list is accessible. Note that this hypothesis is equal to the lemma `elements_inW_implies_multiset_inW` from `Multisets.v`. The lemma from `Multisets.v` isn't applied directly, because when applying this lemma, as a consequence all hypotheses from `Multisets.v` have to be proven.

The lemma `access_between_list_and_WTL` states that when a list of terms is equal to the first projection of a `WTL` converted to a list and the list of terms is accessible with respect to `gt_M`, that then the `WTL` is accessible with respect to `gt_WTL`. The lemma `access_between_proj_and_WTL` states that when the first projection of an `WTL` converted to a list of terms is accessible with respect to `gt_M`, then the `WTL` is accessible with respect to `gt_WTL`.

The third component is the set of terms, ordered with `gt_term_length`. In Coq this is formalized as:

```
Inductive gt_term_length: term->term->Prop:=
  gttermlength: (s,t:term)(gt (length_term s) (length_term t))->
                (gt_term_length s t).
```

Because the well-foundedness of the set of terms ordered with `gt_term_length` translates to well-foundedness of the natural numbers, well-foundedness is added as an hypothesis, named `wflength`.

Now the discussion of the formalization of the proof of well-foundedness of RPO in Coq starts. Go to the last theorem in the Coq file `RPO.v` to follow this discussion. Theorem 7.12 corresponds to `rpo_wf`. Theorem 7.13 corresponds to unfolding `wellfounded` in `rpo_wf`. The proof proceeds by induction on the structure of terms. Note that the base case isn't considered, because only ground terms are formalized in Coq. Because the terms are defined by means of mutual induction, the induction scheme `term_induction` is used here. (See also Section 2.1.) In the second and third clauses of the scheme the lemma's `nilterm_wellfounded` and `wf_in_lists` are used. The first clause of the induction scheme uses `KeyLemma`, which corresponds to Lemma 7.14 exactly. `KeyLemma` is proven from the definition of accessibility and `KeyLemma2`, which corresponds to Lemma 7.15.

`KeyLemma2` is proven by well-founded induction on the tripels. This well-founded induction is indeed allowed because all three components of the tripel are well-founded as is shown above. The formalization in Coq exactly follows the proof given in the thesis and it's annotated in the Coq file in such a way that the structure of proof is recognized. `KeyLemma2` uses only one hypothesis and that is again a hypothesis from the file `Multisets.v`. The hypothesis `remove_add` states that removing an element from a multiset and then adding the same element again leaves the multiset unchanged. At two places of the proof lemma `Aux` is used, this lemma corresponds exactly to Lemma 7.16. Note that in the call of `Aux` in `KeyLemma2` some hypotheses are given to ensure all hypotheses of `Aux` are fullfilled. The proof of `Aux` follows the proof on paper exactly and is also annotated in such a way that the structure of the proof is recognized.

Chapter 8

Conclusions and further research

We have presented a formal proof of well-foundedness of RPO in the in proof checker Coq.

The proof doesn't use Kruskal's tree theorem. It is a constructive proof in the style of the Tait and Girard method; it is shown by induction on the definition of terms that all terms are well-founded with respect to RPO.

We consider the multiset version of RPO [Der82] in a refined form: instead of the multiset ordering we use multiset reduction as in [Nip98], where a multiset M is greater than a multiset N if N is obtained from M by replacing one element with a finite number of smaller elements. For well-foundedness is it sufficient to consider this variant.

In the formalization we use the following hypotheses:

1. Equality on terms is decidable.
2. Well-foundedness of the natural numbers.
3. Well-foundedness of the length of finite first-order terms ordered with $>$.
4. A finite multiset of terms that are well-founded with respect to RPO is well-founded with respect to the multiset reduction of RPO.

We expect the first hypothesis to be easy to prove. The second one is in fact in the Coq library only in a slightly different form. So it needs to be adapted. The third one follows from well-foundedness of the natural numbers. The fourth one is a special case of the theorem stating that a finite multiset is well-founded if all its elements are well-founded. This theorem is proven in the file `Multisets.v`, but because that file still contains quite some hypotheses concerning calculations with multisets, the theorem couldn't be easily applied in the file `RPO.v`. Hence instead we use the application of the theorem as a hypothesis.

Now we discuss to what extent the goals presented in the Introduction are realized. The first goal of this thesis is realized: The original definition of RPO due to Dershowitz[Der82] is formalized in Coq. The formalization follows the definition closely. Nevertheless it wasn't possible to extensively check this formalization. Especially Example 7.5 is suitable for checking the formalization. Don't forget that RPO is formalized for ground terms only and that it uses the multiset reduction instead of multiset ordering. An idea for further research is to fully check the formalization of RPO. For instance a tactic could be written to determine if two given terms are RPO related. To formalize RPO for open terms is also possible.

The second goal isn't realized at all. That RPO is a strict partial order wasn't possible to formalize, because we chose to formalize the multiset reduction instead of the multiset order. The formalized RPO isn't transitive, because it uses the multiset reduction, which isn't transitive either. Reflexivity couldn't be formalized either, but this was due to formalization of multisets,

which is still full of hypotheses.

The third goal is realized under certain hypotheses: The proof of well-foundedness of RPO from [JR99] is formalized in Coq.

From the goals which were added later the first goal is realized: the lexicographic order of the tripels, the proof of it being a strict partial order and the proof of well-foundedness of the lexicographic order are formalized.

The second added goal isn't completely realized, since the file `Multisets.v` still contains many hypotheses concerning calculating with multisets. A direction for research is to try to formalize the multisets extensively. Formalize the multisets so extensively that calculating with multisets is possible and that there are no hypotheses left. Formalize explicit multisets and formalize some examples from the thesis. That the multiset reduction is a strict partial order shouldn't be missing in the formalization. A formalization of the multiset order instead of the multiset reduction would also be usefull, because then maybe RPO is a strict partial order can be formalized.

Although not all goals are realized, the formalization of RPO and the proof of well-foundedness of RPO is done in Coq. A big advantage is the annotation in the Coq files which make the Coq code more transparant and understandable. The thesis should be a good reference in starting with further research. We think that the most interesting direction for further research is to try to solve the above problems to contentment and then for instance try to formalize higher-order RPO (HORPO). Then use the proof of Jouannaud and Rubio[JR99] to formalize the proof of HORPO being well-founded. Or try to formalize the other sorts of RPO which are present, like RPO which uses the lexicographic order, due to Kamin and Levy[KL80].

Bibliography

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[CH88] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Coq03] The development team of Coq. The Coq proof asssitant reference manual, version 7.4, feb 2003. URL: `http://pauillac.inria.fr/coq/`.

[CPM89] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog '88*, volume 417 of *LNCS*. Springer-Verlag, 1989.

[Der82] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1):69—116, 1987.

[DJ90] N. Dershowitz and J-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 243–320. Elsevier, Amsterdam, 1990.

[Fer95] M.C.F. Ferreira. *Termination of term rewriting.* PhD thesis, Utrecht University, Utrecht, The Netherlands, 1995.

[GL01] J. Goubault-Larrecq. Well-founded recursive relations. In L. Fribourg, editor, *Proceedings of the 15th International Workshop on Computer Science Logic (CSL 2001)*, number 2142 in LNCS, pages 484–497, Paris, France, September 2001. Springer Verlag.

[HKPM02] G. Huet, G. Kahn, and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial, version 7.4, 2002. URL: `http://pauillac.inria.fr/coq/`.

[JR99] 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.

[JR00] J-P. Jouannaud and A. Rubio. Higher-order recursive path orderings à la carte. URL: `http://www.lix.polytechnique.fr/~jouannaud/articles/uniformhorpo.pdf`, 2000.

[KL80] S. Kamin and J-J. Levy. Attempts for generalizing the recursive path ordering. Unpublished manuscript, 1980.

[Lec95] F. Leclerc. Termination proof of term rewriting system 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), number 902 in LNCS, pages 312–327, Edinburgh, Scotland, April 1995. Springer Verlag.

[Nip98] T. Nipkow. An inductive proof of the well-foundedness of the multiset order. A proof due to W. Buchholz, URL:`http://www4.informatik.tu-muenchen.de/~nipkow/misc/index.html`, 1998.

[Pau86] L. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[Per03] H. Persson. Constructive termination of recursive path relations. URL: `http://www.math.chalmers.se/~henrikp/`, 2003.

[Raa01] 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. Long version available via `http://www.cs.vu.nl/~femke`.

[Ter03] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

Appendix A: Theorems of the n-fold lexicographic order

Theorem 8.1 Let $S_1, \ldots, S_n$ be n sets ordered with strict partial orders $>_{S_1}, \ldots, >_{S_n}$. The lexicographic order $>_{lex}$ on $S_1 \times \ldots \times S_n$ is again a strict partial order.

Proof To prove that $>_{lex}$ is irreflexive and transitive. The proof proceeds by induction on n.

Base case Suppose $n = 1$. Then we have the lexicographic order of 1 set and that is exactly the set itself. And this set is a strict partial order by assumption.

Induction step From the induction hypothesis that the lexicographic order of $S_2, \ldots, S_n$ is a strict partial order, we want to prove that the lexicographic order of $S_1, \ldots, S_n$ is a strict partial order. Suppose we have the product $S_1 \times \ldots \times S_n$. We can see this product as the product of $S_1 \times (S_2 \times \ldots \times S_n)$, because of the associativity of the product. First prove irreflexivity and then transitivity.

Irreflexive From the induction hypothesis $[(s_2, \ldots, s_n) >_{lex} (s_2, \ldots, s_n)] \Rightarrow false$, we want to prove that $[(s_1, \ldots, s_n) >_{lex} (s_1, \ldots, s_n)] \Rightarrow false$.
Suppose $(s_1, \ldots, s_n) >_{lex} (s_1, \ldots, s_n)$ holds. By the definition of the lexicographic product two cases are possible: $s_1 >_{S_1} s_1$ or ($s_1 =_{S_1} s_1$ and $(s_2, \ldots, s_n) >_{lex} (s_2, \ldots, s_n)$). The first case implies false, because $>_{S_1}$ is irreflexive, so $s_1 >_{S_1} s_1$ is impossible. The second case implies false because of the induction hypothesis. Now irreflexivity is proven.

Transitive From the induction hypothesis that $S_2 \times \ldots \times S_n$ is transitive, we want to prove that $S_1 \times \ldots \times S_n$ is transitive. Let $(x_1, \ldots, x_n) >_{lex} (y_1, \ldots, y_n)$ and $(y_1, \ldots, y_n) >_{lex} (z_1, \ldots, z_n)$ be given with $(x_1, \ldots, x_n)$, $(y_1, \ldots, y_n)$ and $(z_1, \ldots, z_n)$ from $S_1 \times \ldots \times S_n$. We want to prove that holds $(x_1, \ldots, x_n) >_{lex} (z_1, \ldots, z_n)$. From $(x_1, \ldots, x_n) >_{lex} (y_1, \ldots, y_n)$ follows that:

$x_1 > y_1$ or
$x_1 = y_1$ and $(x_2, \ldots, x_n) >_{lex} (y_2, \ldots, y_n)$

From $(y_1, \ldots, y_n) >_{lex} (z_1, \ldots, z_n)$ follows that:

$y_1 > z_1$ or
$y_1 = z_1$ and $(y_2, \ldots, y_n) >_{lex} (z_2, \ldots, z_n)$

Combining these possibilities we have four cases, for which we all can prove that $(x_1, \ldots, x_n) >_{lex} (z_1, \ldots, z_n)$ holds.

1. $x_1 > y_1$ and $y_1 > z_1$ implies $x_1 > z_1$ by the transitivity of $>$. From $x_1 > z_1$ follows by the definition of the lexicographic order that $(x_1, \ldots, x_n) >_{lex} (z_1, \ldots, z_n)$.
2. $x_1 > y_1$ and $[y_1 = z_1$ and $(y_2, \ldots, y_n) >_{lex} (z_2, \ldots, z_n)]$ implies $x_1 > z_1$ by replacing y_1 by z_1. From $x_1 > z_1$ follows by the definition of the lexicographic order that $(x_1, \ldots, x_n) >_{lex} (z_1, \ldots, z_n)$.

3. $[x_1 = y_1$ and $(x_2, \ldots, x_n) >_{lex} (y_2, \ldots, y_n)]$ and $y_1 > z_1$ implies $x_1 > z_1$ by replacing y_1 by x_1. From $x_1 > z_1$ follows by the definition of the lexicographic order that $(x_1, \ldots, x_n) >_{lex} (z_1, \ldots, z_n)$.
4. $(x_1 = y_1$ and $(x_2, \ldots, x_n) >_{lex} (y_2, \ldots, y_n))$ and
   $(y_1 = z_1$ and $(y_2, \ldots, y_n) >_{lex} (z_2, \ldots, z_n))$ implies
   $(x_1 = z_1$ and $(x_2, \ldots, x_n) >_{lex} (z_2, \ldots, z_n))$ by replacing y_1 by z_1 and the transitivity of $>_{lex}$ on $S_2 \times \ldots \times S_n$ (induction hypothesis).
   From $[x_1 = z_1$ and $(x_2, \ldots, x_n) >_{lex} (z_2, \ldots, z_n)]$ follows by the definition of the lexicographic order that $(x_1, \ldots, x_n) >_{lex} (z_1, \ldots, z_n)$.

Theorem 8.2 A lexicographic order of n sets which are all ordered by a well-founded strict partial order is again well-founded.

Proof Suppose we have n sets $S_1, \ldots, S_n$ ordered by well-founded strict partial orders $>_{S_1}, \ldots, >_{S_n}$. We want to prove that the lexicographic order of the product $(S_1 \times \ldots \times S_n)$ is again well-founded. The proof proceeds by induction on n.

Base case Suppose $n = 1$. Then we have the lexicographic product of 1 set, and that is exactly the set itself. And this set is well-founded by assumption.

Induction step Suppose $n > 1$. We have to prove from $(S_2 \times \ldots \times S_n)$ and S_1 is well-founded, that $(S_1 \times \ldots \times S_n)$ is well-founded. We can see this product as the product of $(S_1 \times (S_2 \times \ldots \times S_n))$, because of the associativity of the product. The thing we want to prove now is $\forall s_1 \in S_1 \;\; \forall (s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n : (s_1, s_2, \ldots, s_n) \in W_{S_1 \times \ldots \times S_n}$. This is stated in the following lemma

Lemma 8.3 $\forall s_1 \in S_1 \;\; \forall (s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n : (s_1, s_2, \ldots, s_n) \in W_{S_1 \times \ldots \times S_n}$.

Proof This proof proceeds by well-founded induction on s_1. Remark that this is possible because S_1 is well-founded. To prove is the conclusion of applying well-founded induction with $P(s_1) = \forall (s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n : (s_1, s_2, \ldots, s_n) \in W_{S_1 \times \ldots \times S_n}$. If $\forall s_1 \in S_1 (\forall s_1' \in S_1 : s_1 > s_1' \Rightarrow P(s_1')) \Rightarrow P(s_1)$ holds, we can apply well-founded induction and then this lemma is proven. This is stated in the following lemma:

Lemma 8.4 $\forall s_1 \in S_1 (\forall s_1' \in S_1 : s_1 > s_1' \Rightarrow P(s_1')) \Rightarrow P(s_1)$.

Proof First assume that we have an arbitrary $s_{1_0} \in A$ and that we have $\forall s_1' \in S_1 : s_1 > s_1' \Rightarrow P(s_1')$. We need to show that $P(s_{1_0})$ holds. Remember that $P(s_{1_0}) = \forall (s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n : (s_{1_0}, s_2, \ldots, s_n) \in W_{S_1 \times \ldots \times S_n}$. The rest of this proof proceeds by well-founded induction on $(s_2, \ldots, s_n)$ with $P((s_2, \ldots, s_n)) = (s_{1_0}, s_2 \ldots, s_n) \in W_{S_1 \times \ldots \times S_n}$. If $\forall (s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n (\forall (s_2', \ldots, s_n') \in S_2 \times \ldots \times S_n : (s_2, \ldots, s_n) > (s_2', \ldots, s_n') \Rightarrow P(b')) \Rightarrow P(b)$ holds, we can apply well-founded induction and then this lemma is proven. This is stated in the following lemma:

Lemma 8.5 $\forall (s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n (\forall (s_2', \ldots, s_n') \in S_2 \times \ldots \times S_n : (s_2, \ldots, s_n) > (s_2', \ldots, s_n') \Rightarrow P(b')) \Rightarrow P(b)$.

Proof First assume that we have an arbitrary $(s_{2_0}, \ldots, s_{n_0}) \in S_2 \times \ldots \times S_n$ and that we have $\forall (s_2', \ldots, s_n') \in S_2 \times \ldots \times S_n : (s_2, \ldots, s_n) > (s_2', \ldots, s_n') \Rightarrow P(b')$. We need show that $P((s_{2_0}, \ldots, s_{n_0}))$ holds. Remember that $P((s_{2_0}, \ldots, s_{n_0})) = (s_{1_0}, s_{2_0}, \ldots, s_{n_0}) \in W_{S_1 \times \ldots \times S_n}$. We have $(s_{1_0}, s_{2_0}, \ldots, s_{n_0}) \in W_{S_1 \times \ldots \times S_n}$ if we can prove that for all $(t_1, \ldots, t_n)$ with $(s_{1_0}, s_{2_0}, \ldots, s_{n_0}) >_{lex} (t_1, \ldots, t_n)$ holds that $(t_1, \ldots, t_n)$ is in $W_{S_1 \times \ldots \times S_n}$.
Now $(s_{1_0}, s_{2_0}, \ldots, s_{n_0}) >_{lex} (t_1, \ldots, t_n)$ comes from two cases:

$s_{1_0} > t_1$. In the proof of Lemma 8.4 we assumed that $\forall s'_1 \in S_1 : s_1 > s'_1 \Rightarrow P(s'_1)$ holds. The t_1 we've got here is such an s'_1 and we conclude that $P(t_1) = \forall(s_2, \ldots, s_n) \in S_2 \times \ldots \times S_n : (t_1, s_2, \ldots, s_n) \in W_{S_1 \times \ldots \times S_n}$ holds. Hence follows that $(t_1, t_2, \ldots, t_n) \in W_{S_1 \times \ldots \times S_n}$.

$s_{1_0} = t_1 \wedge (s_{2_0}, \ldots, s_{n_0}) >_{lex} (t_2, \ldots, t_n)$. We assumed that $\forall(s'_2, \ldots, s'_n) \in S_2 \times \ldots \times S_n : (s_2, \ldots, s_n) > (s'_2, \ldots, s'_n) \Rightarrow P(b')$ holds. The $(t_2, \ldots, t_n)$ we've got here is such an $(s'_2, \ldots, s'_n)$ and we conclude that $P((t_2, \ldots, t_n)) = (s_{1_0}, t_2, \ldots, t_n) \in W_{S_1 \times \ldots \times S_n}$ holds. Because $s_{1_0} = t_1$ follows that $(t_1, t_2, \ldots, t_n) \in W_{S_1 \times \ldots \times S_n}$.

Summarizing this proof gives that because Lemma 8.5 is proven, applying well-founded induction two times proves Lemma 8.3, which is the induction case of Theorem 8.2.

Appendix B: Coq file Terms.v

```
(* This file is about ground terms *)
(* it contains 3 sections *)

(* terms: Here ground terms are defined, root,
   subterms, length and an induction scheme are defined too *)

(* termlist_and_list: Gives conversions between a termlist and a list of
   terms and defines when an element is in a termlist *)

(* examples_terms: gives examples of ground terms and their lengths *)

Require PolyList.

Section terms.

(* definition 2.5: definition of a ground term *)
Inductive term : Set :=
  buildterm : nat -> termlist -> term
with termlist : Set :=
  nilterm  : termlist
| consterm : term -> termlist -> termlist .

(* defines an induction scheme for terms *)
Scheme
  term_induction := Induction for term Sort Prop
with
  termlist_induction := Induction for termlist Sort Prop.

Check term_induction.

(* gives the root of a ground term *)
Definition root [s:term] : nat :=
Cases s of
  (buildterm f ss) => f
end.

(* gives the proper subterms of a ground term *)
Definition subterms [s:term] : termlist :=
Cases s of
  (buildterm f ss) => ss
end.
```

```
(* definition 2.8: gives the length of a ground term *)
Fixpoint length_term [s:term] : nat :=
Cases s of
  (buildterm f ss) => (S (length_termlist ss))
end
with
length_termlist [ts:termlist] : nat :=
Cases ts of
  nilterm => O
| (consterm h t) => (plus (length_term h) (length_termlist t))
end .

End terms.


Section termlist_and_list .

(* converts a termlist to a list of terms *)
Fixpoint termlist_to_list [ss:termlist] : (list term) :=
Cases ss of
  nilterm => (nil term)
| (consterm h t) => (cons h (termlist_to_list t))
end .

(* converts a list of terms to a termlist *)
Fixpoint list_to_termlist [ss: (list term)] : termlist :=
Cases ss of
  nil => nilterm
| (cons h t) => (consterm h (list_to_termlist t))
end .

(* determines when an element is in a termlist *)
(* an element is in a termlist if it is in the list of terms *)
Definition in_termlist [s:term;ts:termlist] : Prop :=
(In s (termlist_to_list ts)).

End termlist_and_list .

Section example_terms.

(* example 2.7 *)
(* constant 0*)
Check (buildterm O nilterm).
(* term 0(0) *)
Check (buildterm O (consterm (buildterm O nilterm) nilterm )).
(* term 0(0,0) *)
Check (buildterm O (consterm
        (buildterm O nilterm)
(consterm (buildterm O nilterm) nilterm))).

(* example 2.10 *)
(* length's of above terms *)
Eval Compute in  (length_term (buildterm O nilterm)).
```

```
Eval Compute in  (length_term (buildterm 0
                              (consterm (buildterm 0 nilterm) nilterm ))).

Eval Compute in (length_term (buildterm 0 (consterm (buildterm 0 nilterm)
                             (consterm (buildterm 0 nilterm) nilterm)))).


End example_terms.
```

Appendix C: Coq file Ordering.v

```
(* This file is about orderings and contains three sections. *)
(* Ordering: Definition of orderings and their properties *)
(* Example 1: nat has a total order *)
(* Example 2: some examples of orders *)

Section Ordering.

(* introduce set*)
Variable X:Set.

(* introduce relation*)
Definition relation := X -> X -> Prop.
Variable R: relation.

(* definition 3.1: properties of relations*)
Definition reflexive : Prop := (x: X) (R x x).
Definition transitive : Prop := (x,y,z: X) (R x y) -> (R y z) -> (R x z).
Definition symmetric : Prop := (x,y: X) (R x y) -> (R y x).
Definition antisymmetric : Prop := (x,y: X) (R x y) -> (R y x)-> x=y.
Definition equivalence := reflexive /\ transitive /\ symmetric.
Definition irreflexive : Prop := (x: X) ~(R x x).

(*definition 3.2: definitions of orderings*)
Definition quasiorder: Prop:= reflexive /\ transitive.
Definition strictorder: Prop:= irreflexive /\ transitive.
Definition order: Prop:= reflexive /\ transitive /\ antisymmetric.

(*definition 3.3: properties of orderings*)
Definition totalorder: Prop:= (x,y:X)(R x y)\/(R y x)\/(x=y).
Definition partialorder: Prop:=~(totalorder).

End Ordering.

(*EXAMPLES*)
Section Example1.

Require Arith.

(* example 3.4 *)
(*Proof that natural numbers have total ordering from library*)
(* Module Coq.Arith.Lt *)
Theorem nat_total_order: (m,n: nat) ~ m = n -> (lt m n) \/ (lt n m).
Proof.
```

```
Intros.
Elim(le_or_lt n m).
Intros.
Elim (le_lt_or_eq n m).
Auto.
Intros.
Elim H.
Auto.
Exact H0.
Auto.
Qed.

End Example1.

Section Example2.

(* example 3.5 *)
(* The Set X *)
Inductive Sigma:Set:=
Nul:Sigma | Suc:Sigma | M: Sigma | A:Sigma.


(* Relation 1*)
Inductive R1: Sigma->Sigma-> Prop:=
 R1mo: (R1 M Nul)
|R1ms: (R1 M Suc)
|R1ma: (R1 M A)
|R1as: (R1 A Suc).

Lemma R1strictpartialorder: (strictorder Sigma R1)/\(partialorder Sigma R1).
Proof.
Split. Unfold strictorder. Split. Unfold irreflexive. Intro. Unfold not.
Intro. Inversion H. Unfold transitive. Intros x y z. Intros.
Inversion H; Inversion H0. Apply R1mo. Apply R1ms. Apply R1ma. Apply R1ms.
Apply R1mo. Apply R1ms. Apply R1ma. Apply R1ms. Apply R1mo. Apply R1ms.
Apply R1ma. Apply R1ms. Rewrite <- H2 in H0. Rewrite <- H4 in H0.
Inversion H0. Apply R1as. Rewrite <- H1 in H. Rewrite <- H3 in H.
Inversion H. Apply R1as. Unfold partialorder. Unfold not. Unfold totalorder.
Intro. Cut (R1 Suc Nul)\/(R1 Nul Suc)\/Suc=Nul. Intro. Elim H0. Intro.
Inversion H1. Intro. Elim H1. Intro. Inversion H2. Intro. Inversion H2.
Apply H with x:=Suc y:=Nul.
Qed.

(* Relation 2*)
Inductive R2: Sigma->Sigma-> Prop:=
 R2mo: (R2 M Nul)
|R2ms: (R2 M Suc)
|R2ma: (R2 M A)
|R2as: (R2 A Suc)
|R2mm: (R2 M M)
|R2ss: (R2 Suc Suc)
|R2aa: (R2 A A)
|R2oo: (R2 Nul Nul).
```

```
Lemma R2partialorder: (order Sigma R2)/\(partialorder Sigma R2).
Proof.
Split. Unfold order. Split. Unfold reflexive. Intro. Case x. Apply R2oo.
Apply R2ss. Apply R2mm. Apply R2aa. Split. Unfold transitive. Intros.
Inversion H; Inversion H0. Apply R2mo. Apply R2ms. Apply R2ma. Apply R2ms.
Apply R2mm. Apply R2ms. Apply R2ma. Apply R2mo. Apply R2mo. Apply R2ms.
Apply R2ma. Apply R2ms. Apply R2mm. Apply R2ms. Apply R2ma. Apply R2mo.
Apply R2mo. Apply R2ms. Apply R2ma. Apply R2ms. Apply R2mm. Apply R2ms.
Apply R2ma. Apply R2mo. Rewrite <- H2 in H0. Rewrite <- H4 in H0.
Inversion H0. Apply R2as. Apply R2aa. Apply R2as. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Apply R2as. Apply R2aa. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Apply R2mo. Apply R2ms. Apply R2ma.
Apply R2ms. Apply R2mm. Apply R2ms. Apply R2ma. Apply R2mo.
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Apply R2ss.
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Apply R2ss.
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Apply R2ss.
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Rewrite <- H2 in H0. Rewrite <- H4 in H0.
Inversion H0. Apply R2as. Apply R2aa. Apply R2as. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Apply R2as. Apply R2aa. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Apply R2oo. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Rewrite <- H2 in H0. Rewrite <- H4 in H0.
Inversion H0. Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0.
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Rewrite <- H2 in H0. Rewrite <- H4 in H0.
Inversion H0. Apply R2oo. Unfold antisymmetric. Intros. Inversion H.
Rewrite <- H1 in H0. Rewrite <- H2 in H0. Inversion H0. Rewrite <- H1 in H0.
Rewrite <- H2 in H0. Inversion H0. Rewrite <- H1 in H0. Rewrite <- H2 in H0.
Inversion H0. Rewrite <- H1 in H0. Rewrite <- H2 in H0. Inversion H0.
Auto. Auto. Auto. Auto. Unfold partialorder. Unfold not. Unfold totalorder.
Intro. Cut (R2 Suc Nul)\/(R2 Nul Suc)\/Suc=Nul. Intro. Elim H0. Intro.
Inversion H1. Intro. Elim H1. Intro. Inversion H2. Intro. Inversion H2.
Apply H with x:=Suc y:=Nul.
Qed.

(* Relation 3*)
Inductive R3: Sigma->Sigma-> Prop:=
 R3mo: (R3 M Nul)
|R3ms: (R3 M Suc)
|R3ma: (R3 M A)
|R3as: (R3 A Suc)
|R3ao: (R3 A Nul)
|R3so: (R3 Suc Nul).


Lemma R3stricttotalorder: (strictorder Sigma R3)/\(totalorder Sigma R3).
Proof.
Split. Unfold strictorder. Split. Unfold irreflexive. Intro. Unfold not.
Intro. Inversion H. Unfold transitive. Intros. Inversion H; Inversion H0.
Apply R3mo. Apply R3ms. Apply R3ma. Apply R3ms. Apply R3mo. Apply R3mo.
Apply R3mo. Apply R3ms. Apply R3ma. Apply R3ms. Apply R3mo. Apply R3mo.
Apply R3mo. Apply R3ms. Apply R3ma. Apply R3ms. Apply R3mo. Apply R3mo.
Apply R3ao. Apply R3as. Rewrite <- H2 in H0. Rewrite <- H4 in H0.
Inversion H0. Apply R3as. Apply R3ao. Apply R3ao. Apply R3ao. Apply R3as.
```

```
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Apply R3as. Apply R3ao.
Apply R3ao. Apply R3so. Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0.
Rewrite <- H2 in H0. Rewrite <- H4 in H0. Inversion H0. Rewrite <- H2 in H0.
Rewrite <- H4 in H0. Inversion H0. Apply R3so. Apply R3so. Unfold totalorder.
Intros. Elim x. Elim y. Right. Right. Auto. Right. Left. Apply R3so. Right.
Left. Apply R3mo. Right. Left. Apply R3ao. Elim y. Left. Apply R3so. Right.
Right. Auto. Right. Left. Apply R3ms. Right. Left. Apply R3as. Elim y. Left.
Apply R3mo. Left. Apply R3ms. Right. Right. Auto. Left. Apply R3ma. Elim y.
Left. Apply R3ao. Left. Apply R3as. Right. Left. Apply R3ma. Right. Right.
Auto.
Qed.

(* Relation 4 *)
Inductive R4: Sigma->Sigma-> Prop:=
 R4mo: (R4 M Nul)
|R4ms: (R4 M Suc)
|R4ma: (R4 M A)
|R4as: (R4 A Suc)
|R4ao: (R4 A Nul)
|R4so: (R4 Suc Nul)
|R4mm: (R4 M M)
|R4ss: (R4 Suc Suc)
|R4aa: (R4 A A)
|R4oo: (R4 Nul Nul).

Lemma R4totalorder: (order Sigma R4)/\(totalorder Sigma R4).

Proof.
Split. Unfold order. Split. Unfold reflexive. Intro. Case x. Apply R4oo.
Apply R4ss. Apply R4mm. Apply R4aa. Split. Unfold transitive. Intros.
Inversion H;Inversion H0. Apply R4mo. Apply R4ms. Apply R4ma. Apply R4ms.
Apply R4mo. Apply R4mo. Apply R4mm. Apply R4ms. Apply R4ma. Apply R4mo.
Apply R4mo. Apply R4ms. Apply R4ma. Apply R4ms. Apply R4mo. Apply R4mo.
Apply R4mm. Apply R4ms. Apply R4ma. Apply R4mo. Apply R4mo. Apply R4ms.
Apply R4ma. Apply R4ms. Apply R4mo. Apply R4mo. Apply R4mm. Apply R4ms.
Apply R4ma. Apply R4mo. Apply R4ao. Apply R4as. Apply R4aa. Apply R4as.
Apply R4ao. Apply R4ao. Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0.
Apply R4as. Apply R4aa. Apply R4ao. Apply R4ao. Apply R4as. Apply R4aa.
Apply R4as. Apply R4ao. Apply R4ao. Rewrite <-H2 in H0. Rewrite <-H4 in H0.
Inversion H0. Apply R4as. Apply R4aa. Apply R4ao. Apply R4so. Apply R4ss.
Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Apply R4ss. Apply R4so.
Apply R4so. Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Apply R4ss.
Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Apply R4so. Apply R4mo.
Apply R4ms. Apply R4ma. Apply R4ms. Apply R4mo. Apply R4mo. Apply R4mm.
Apply R4ms. Apply R4ma. Apply R4mo. Apply R4so. Apply R4ss. Rewrite <-H2 in H0.
Rewrite <-H4 in H0. Inversion H0. Apply R4ss. Apply R4so. Apply R4so.
Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Apply R4ss.
Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Apply R4so. Apply R4ao.
Apply R4as. Apply R4aa. Apply R4as. Apply R4ao. Apply R4ao. Rewrite <-H2 in H0.
Rewrite <-H4 in H0. Inversion H0. Apply R4as. Apply R4aa. Apply R4ao.
Apply R4oo. Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0.
Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Rewrite <-H2 in H0.
Rewrite <-H4 in H0. Inversion H0. Apply R4oo. Apply R4oo. Rewrite <-H2 in H0.
Rewrite <-H4 in H0. Inversion H0. Rewrite <-H2 in H0. Rewrite <-H4 in H0.
```

```
Inversion H0. Rewrite <-H2 in H0. Rewrite <-H4 in H0. Inversion H0. Apply R4oo.
Unfold antisymmetric. Intros. Inversion H. Rewrite <- H1 in H0.
Rewrite <- H2 in H0. Inversion H0. Rewrite <- H1 in H0. Rewrite <- H2 in H0.
Inversion H0. Rewrite <- H1 in H0. Rewrite <- H2 in H0. Inversion H0.
Rewrite <- H1 in H0. Rewrite <- H2 in H0. Inversion H0. Rewrite <- H1 in H0.
Rewrite <- H2 in H0. Inversion H0. Rewrite <- H1 in H0. Rewrite <- H2 in H0.
Inversion H0. Auto. Auto. Auto. Auto. Unfold totalorder. Intros. Elim x.
Elim y. Left. Apply R4oo. Right. Left. Apply R4so. Right. Left. Apply R4mo.
Right. Left. Apply R4ao. Elim y. Left. Apply R4so. Left. Apply R4ss. Right.
Left. Apply R4ms. Right. Left. Apply R4as. Elim y. Left. Apply R4mo. Left.
Apply R4ms. Left. Apply R4mm. Left. Apply R4ma. Elim y. Left. Apply R4ao. Left.
Apply R4as. Right. Left. Apply R4ma. Left. Apply R4aa.
Qed.

End Example2.
```

Appendix D: Coq file Wellfounded.v

```
(* This file contains exactly the same clauses as the module Coq.Init.Wf,
   except that the orientation of the relation on A is changed.
   We oriented the relation as a 'greater than' relation instead
   of a 'smaller than' relation. *)

(* The section about Fixpoint isn't added because we don't need it*)

(* This file contains three sections *)
(* def_Wf: definitions of access and wellfounded*)
(* wf_ind: definitions of well-founded induction *)
(* wf_part_ind: definitions of well-founded part induction*)

Section def_Wf.

Require Logic.
Require LogicSyntax.

Variable A : Set.
Variable gtA : A -> A -> Prop.

(* definition 4.1: definition of accessible *)
Inductive Access : A -> Prop :=
  Access_intro : (x:A)((y:A)(gtA x y)->(Access y))->(Access x).

(* note after definition 4.1 *)
(* this lemma states that if an element is accessible, then all smaller
   elements are accessible too. *)
Lemma Access_inv : (x:A)(Access x) -> (y:A)(gtA x y) -> (Access y).

Proof.
NewDestruct 1; Trivial.
Defined .

Section AccessRecType.
  Variable P : A -> Type.
  Variable F : (x:A)((y:A)(gtA x y)->(Access y))->
               ((y:A)(gtA x y)->(P y))->(P x).

Fixpoint Access_rect [x:A;a:(Access x)] : (P x)
     := (F x (Access_inv x a) ([y:A][h:(gtA x y)]
```

```
        (Access_rect y (Access_inv x a y h) ))).

End AccessRecType.

Definition Access_rec [P:A->Set] := (Access_rect P).

(*definition 4.2: definition of well-founded *)
Definition wellfounded := (a:A)(Access a).

End def_Wf.

Section wf_ind.

Variable A : Set.
Variable gtA : A -> A -> Prop.

Theorem wf_induction_type :
  (P:A->Type)
  (wellfounded A gtA)->
  ((x:A)((y:A)(gtA x y)->(P y))->(P x))
  ->
  (a:A)(P a).

Proof.
Intros.
Unfold wellfounded in H.
Elim H with a.
Intros.
Apply X.
Exact X0.
Qed.

Theorem wf_induction :
  (P:A->Set)
  (wellfounded A gtA)->
  ((x:A)((y:A)(gtA x y)->(P y))->(P x))
  ->
  (a:A)(P a).

Proof.
Intro.
Apply wf_induction_type.
Qed.

(* definition 4.4: well-founded induction principle *)
Theorem wf_ind :
  (P:A->Prop)
  (wellfounded A gtA) ->
  ((x:A)((y:A)(gtA x y)->(P y))->(P x))
  ->
  (a:A)(P a).

Proof.
Intro.
```

```
Apply wf_induction_type.
Qed.

End wf_ind.

Section wf_part_ind.

Variable A : Set.
Variable gtA : A -> A -> Prop.

Theorem wf_part_induction_type:
  (P:A->Type)
  ((x:A)(Access A gtA x) ->
  ((y:A)(gtA x y)->(P y))->(P x))
  ->
  (a:A)(Access A gtA a) -> (P a).

Proof.
Intros .
Elim H .
Intros .

Apply X .
Apply Access_intro .
Assumption .
Assumption .
Qed .

Theorem wf_part_induction:
  (P:A->Set)
  ((x:A) (Access A gtA x) ->
  ((y:A)(gtA x y)->(P y))->(P x))
  ->
  (a:A)(Access A gtA a) -> (P a).

Proof.
Intro.
Apply wf_part_induction_type.
Qed.

(* definition 4.5: well-founded part induction principle *)
Theorem wf_part_ind:
  (P:A->Prop)
  ((x:A) (Access A gtA x) ->
  ((y:A)(gtA x y)->(P y))-> (P x))
  ->
  (a:A)(Access A gtA a) -> (P a).

Proof.
Intro.
Apply wf_part_induction_type .
Qed.

End wf_part_ind.
```

Appendix E: Coq file Lex2.v

```
(* This file is about the 2-fold lexicographic order *)
(* it contains six sections *)
(* Pairs:    Defines independent pairs from A x B *)
(* Example_pairs: Makes some explicit pairs from nat x nat *)
(* Lexord_pairs:  Defines the lexord of two sets *)
(* Example_lexord_pairs: Gives examples of lexord of pairs from nat x nat*)
(* Strictorder:   Proves that lexord of strictorders is a strictorder *)
(* Wellfounded:   Proves that lexord of wellfounded orders is wellfounded *)

Load Ordering.
Load Wellfounded.

Section Pairs.

Variable A : Set .
Variable B : Set .

(* definition of an independent pair from A x B *)
Inductive sigSpair [A:Set;B:Set] : Set :=
  existSpair : (x:A)(y:B)(sigSpair A B) .

(* projections of the pair from A x B *)
Definition projS1pair :=
  [x:(sigSpair A B)]Cases x of (existSpair a _) => a end.

Definition projS2pair :=
  [x:(sigSpair A B)]Cases x of (existSpair _ b) => b end.

End Pairs.

Section Example_pairs .

(* example 5.4 *)
Check (existSpair nat nat) .
Check (existSpair nat nat O O ) .
Check (existSpair nat nat (S O) O).

Check (projS1pair nat nat).
Check (projS1pair nat nat (existSpair nat nat O O)) .
Check (projS1pair nat nat (existSpair nat nat (S O) O)) .

Check (projS2pair nat nat).
Check (projS2pair nat nat (existSpair nat nat O O)) .
```

```
Check (projS2pair nat nat (existSpair nat nat (S O) O)) .

Eval Compute in (projS1pair nat nat (existSpair nat nat (S O) O)) .
Eval Compute in (projS2pair nat nat (existSpair nat nat (S O) O)) .

End Example_pairs .

Section Lexord_pairs .

Variable A : Set .
Variable B : Set .
Variable gtA : A -> A -> Prop.
Variable gtB : B -> B -> Prop .

(* definition 5.2: definition of the 2-fold lexicographic order *)
Inductive lexordpair : (sigSpair A B) -> (sigSpair A B) -> Prop :=
  left_lex_pair :  (a,a':A)(b,b': B)
                   (gtA a a') ->
                   (lexordpair (existSpair A B a b) (existSpair A B a' b'))
| right_lex_pair : (a:A)(b,b':B)
                   (gtB b b') ->
                   (lexordpair (existSpair A B a b) (existSpair A B a b')).

End Lexord_pairs .

Section Example_lexord_pairs.
(* example 5.5 *)

(* Checks *)
Check lexordpair.
Check (lexordpair nat nat).
Check (lexordpair nat nat gt gt ).
Check (lexordpair nat nat gt gt (existSpair nat nat (S O) O)
                                (existSpair nat nat  O (S O))).

(* Lemma (1,0) >lex (0,1) *)
Lemma greater1: (lexordpair nat nat gt gt
                  (existSpair nat nat (S O) O)
                  (existSpair nat nat O (S O))) .

Proof.
Apply left_lex_pair.
Auto.
Qed.

(* Lemma (0,1) >lex (0,0) *)
Lemma greater2: (lexordpair nat nat gt gt
                  (existSpair nat nat  O (S O) )
                  (existSpair nat nat O  O)) .

Proof.
Apply right_lex_pair.
Auto.
Qed.
```

```
Require Le.
(*Lemma (0,1) >lex (1,0) implies false *)
Lemma notgreater1: (lexordpair nat nat gt gt
                     (existSpair nat nat  O (S O)  )
                     (existSpair nat nat (S O) O ))
                   -> False .

Proof.
Intro.
Inversion H.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:=(S O).
Exact H1.
Qed.

(*Lemma (0,0) >lex (0,0) implies false *)
Lemma notgreater2: (lexordpair nat nat gt gt
                     (existSpair nat nat  O O )
                     (existSpair nat nat O O))
                   -> False .

Proof.
Intro.
Inversion H.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:=O.
Exact H1.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:=O.
Exact H1.
Qed.

End Example_lexord_pairs .

Section Strictorder.

Variable A : Set .
Variable B : Set .
Variable gtA : A -> A -> Prop.
Variable gtB : B -> B -> Prop .

(* theorem 5.9 *)
Lemma lexordpair_strictorder :
(strictorder A gtA)->
(strictorder B gtB)->
(strictorder (sigSpair A B) (lexordpair A B gtA gtB)).
```

```
Proof.
Intros strictorderA strictorderB.
Unfold strictorder.
Split.

(* irreflexive *)
Unfold irreflexive.
Intro.
Unfold not.
Intro.
Inversion H.

(* Case left_lex *)
Unfold strictorder in strictorderA.
Elim strictorderA.
Unfold irreflexive.
Intros.
Apply H4 with
x:=a.
Exact H2.

(*Case right_lex *)
Unfold strictorder in strictorderB.
Elim strictorderB.
Unfold irreflexive.
Intros.
Apply H3 with
x:= b.
Exact H2.

(*transitive*)
Unfold transitive.
Intros.

(* Case distinction is made by Inversion *)
Inversion H;
Inversion H0.

(* Case left_lex, a1 > a2 /\ a2 > a3 *)
Apply left_lex_pair.
Rewrite <- H3 in H5.
Cut a0=a'.
Intro.
Rewrite <- H7 in H1.
Unfold strictorder in strictorderA.
Elim strictorderA.
Unfold transitive.
Intros.
Apply H9 with
x:=a
y:=a0
z:=a'0.
Exact H1.
Exact H4.
```

```
Change (projS1pair A B (existSpair A B a0 b0)) =
       (projS1pair A B (existSpair A B a' b')).
Rewrite H5.
Reflexivity.

(* Case left_lex, a1 > a2 /\ (a2=a3 /\ b2> b3) *)
Apply left_lex_pair.
Rewrite <- H3 in H5.
Cut a0=a'.
Intro.
Rewrite <-H7 in H1.
Exact H1.
Change (projS1pair A B (existSpair A B a0 b0)) =
       (projS1pair A B (existSpair A B a' b')).
Rewrite H5.
Reflexivity.

(*Case left_lex, (a1=a2 /\ b1>b2) /\a2>a3 *)
Apply left_lex_pair.
Rewrite <- H3 in H5.
Cut a0=a.
Intro.
Rewrite -> H7 in H4.
Exact H4.
Change (projS1pair A B (existSpair A B a0 b0)) =
       (projS1pair A B (existSpair A B a b')).
Rewrite H5.
Reflexivity.

(* Case right_lex, (a1=a2 /\ b1>b2) /\ (a2=a3 /\b2>b3) *)
Rewrite <- H3 in H5.
Cut a0=a.
Intros.
Rewrite H7.
Apply right_lex_pair.
Cut b'=b0.
Intro.
Unfold strictorder in strictorderB.
Elim strictorderB.
Unfold transitive.
Intros.
Rewrite <- H8 in H4.
Apply H10 with
x:= b
y:= b'
z:= b'0.
Exact H1.
Exact H4.
Apply sym_eq.
Change (projS2pair A B (existSpair A B a0 b0)) =
       (projS2pair A B (existSpair A B a b')).
Rewrite H5.
Reflexivity.
Change (projS1pair A B (existSpair A B a0 b0)) =
```

```
       (projS1pair A B (existSpair A B a b')).
Rewrite H5.
Reflexivity.
Qed.
End Strictorder.

Section Wellfounded.

Variable A : Set .
Variable B : Set .
Variable gtA : A -> A -> Prop.
Variable gtB : B -> B -> Prop .

(* theorem 5.12 *)
Lemma lexordpair_wellfounded :
  (wellfounded A gtA) ->
  (wellfounded B gtB) ->
  (wellfounded (sigSpair A B) (lexordpair A B gtA gtB)) .

Proof .
Unfold wellfounded .
Intro WFA .
Intro WFB .
Induction a .
Rename a into x .

(* lemma 5.13 *)
Intro a .

(* applying well-founded induction on a *)
Apply wf_ind with
  P := [a:A](b:B)(Access (sigSpair A B)
                 (lexordpair A B gtA gtB) (existSpair A B a b))
  gtA := gtA .
Unfold wellfounded .
Apply WFA .

(* lemma 5.14 *)
Intro a0 .
Intro IHA .
Intro b .

(* applying well-founded induction on b *)
Apply wf_ind with
  P := [b:B](Access (sigSpair A B)
            (lexordpair A B gtA gtB) (existSpair A B a0 b))
  gtA := gtB .
Unfold wellfounded .
Apply WFB .

(* lemma 5.15 *)
Intro b0 .
Intro IHB .
```

```
(* using the definition of accessible *)
Apply Access_intro .
Induction y .
Intro c .
Intro d .
Intro H .

(* Case distinction is made by inversion *)
Inversion H .

(* Case a0>c *)
Apply IHA with y := c .
Apply H1 .
Rewrite <- H3.

(* Case a0=c /\ b0 > d *)
Apply IHB with y := d .
Apply H1 .
Qed .

End Wellfounded.
```

Appendix F: Coq file Lex3.v

```
(* This file is about the 3-fold lexicographic order *)

(* It contains six sections *)
(* Tripels:      Defines independent tripels from A x B x C *)
(* Example_tripels: Makes some explicit tripels from nat x nat x nat *)
(* Lexord_tripels: Defines the lexord of three sets *)
(* Example_lexord_tripels: Gives examples of lexord of tripels from
   nat x nat x nat *)
(* Strictorder:     Proves that lexord of strictorders is a strictorder *)
(* Wellfounded:     Proves that lexord of wellfounded orders is wellfounded *)

Load Ordering.
Load Wellfounded.

Section Tripels.

Variable A : Set.
Variable B : Set.
Variable C : Set.

(* definition of an independent tripel from A x B x C *)
Inductive sigStripel[A,B,C:Set]: Set :=
  existStripel : (x:A)(y:B)(z:C)(sigStripel A B C).

(* projections of the tripel from A x B x C *)
Definition projS1tripel :=
  [x:(sigStripel A B C)]Cases x of (existStripel a _ _ ) => a end.

Definition projS2tripel :=
  [x:(sigStripel A B C)]Cases x of (existStripel _ b _ ) => b end.

Definition projS3tripel :=
  [x:(sigStripel A B C)]Cases x of (existStripel _ _ c ) => c end.

End Tripels.

Section Example_tripels.

(* example 5.6 *)
Check (existStripel nat nat nat) .
Check (existStripel nat nat nat O O O ) .
Check (existStripel nat nat nat (S (S O)) (S O) O).
```

```
Check (projS1tripel nat nat nat).

Check (projS1tripel nat nat nat (existStripel nat nat nat (S (S O)) (S O) O)) .
Check (projS2tripel nat nat nat (existStripel nat nat nat (S (S O)) (S O) O)) .
Check (projS3tripel nat nat nat (existStripel nat nat nat (S (S O)) (S O) O)) .

Eval Compute in (projS1tripel nat nat nat (existStripel nat nat nat
                                                    (S (S O)) (S O) O)) .
Eval Compute in (projS2tripel nat nat nat (existStripel nat nat nat
                                                    (S (S O)) (S O) O)) .
Eval Compute in (projS3tripel nat nat nat (existStripel nat nat nat
                                                    (S (S O)) (S O) O)) .

End Example_tripels .

Section Lexord_tripels.

Variable A : Set .
Variable B : Set .
Variable C : Set .

Variable gtA : A -> A -> Prop .
Variable gtB : B -> B -> Prop .
Variable gtC : C -> C -> Prop .

(* definition 5.3: definition of the 3-fold lexicographic order *)
Inductive lexordtripel : (sigStripel A B C) -> (sigStripel A B C) -> Prop :=

  left_lex_tripel :   (a,a':A)(b,b': B)(c,c':C)
                      (gtA a a') ->
                      (lexordtripel (existStripel A B C a b c)
                                    (existStripel A B C a' b' c'))
| middle_lex_tripel : (a:A)(b,b': B)(c,c':C)
                      (gtB b b') ->
                      (lexordtripel (existStripel A B C a b c)
                      (existStripel A B C a b' c'))
| right_lex_tripel :  (a:A)(b:B)(c,c':C)
                      (gtC c c') ->
                      (lexordtripel (existStripel A B C a b c)
                                    (existStripel A B C a b c')).

End Lexord_tripels.

Section Example_lexord_tripels.
(* example 5.7 *)

(* Checks *)
Check lexordtripel.
Check (lexordtripel nat nat nat).
Check (lexordtripel nat nat nat gt gt gt ).
Check (lexordtripel nat nat nat gt gt gt (existStripel nat nat nat (S O) O O)
                                         (existStripel nat nat nat O O (S O))).

(* Lemma (1,1,1) >lex (0,1,1) *)
```

```
Lemma greater1: (lexordtripel nat nat nat gt gt gt
                  (existStripel nat nat nat (S O) (S O) (S O))
                  (existStripel nat nat nat O (S O) (S O))).

Proof.
Apply left_lex_tripel.
Auto.
Qed.

(* Lemma (1,1,1) >lex (1,0,1) *)
Lemma greater2: (lexordtripel nat nat nat gt gt gt
                  (existStripel nat nat nat (S O) (S O) (S O))
                  (existStripel nat nat nat (S O) O (S O))).

Proof.
Apply middle_lex_tripel.
Auto.
Qed.

(* Lemma (1,1,1) >lex (1,1,0) *)
Lemma greater3: (lexordtripel nat nat nat gt gt gt
                  (existStripel nat nat nat (S O) (S O) (S O))
                  (existStripel nat nat nat (S O) (S O) O)).

Proof.
Apply right_lex_tripel.
Auto.
Qed.

Require Le.
(* Lemma (0,0,0) >lex (0,0,0) implies false *)
Lemma notgreater1:(lexordtripel nat nat nat gt gt gt
                    (existStripel nat nat nat O O O)
                    (existStripel nat nat nat O O O))
                  -> False.

Proof.
Intro.
Inversion H.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:=O.
Exact H1.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:=O.
Exact H1.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:=O.
Exact H1.
```

```
Qed.

(* Lemma (0,1,1) >lex (1,0,0) implies false *)
Lemma notgreater2:(lexordtripel nat nat nat gt gt gt
                    (existStripel nat nat nat O (S O) (S O))
                    (existStripel nat nat nat (S O) O O))
                  -> False.

Proof.
Intro.
Inversion H.
Unfold gt in H1.
Unfold lt in H1.
Apply le_Sn_O with
n:= (S O).
Exact H1.
Qed.

End Example_lexord_tripels.

Section Strictorder.

Variable A : Set .
Variable B : Set .
Variable C : Set .
Variable gtA : A -> A -> Prop.
Variable gtB : B -> B -> Prop .
Variable gtC : C -> C -> Prop.

(* theorem 5.10 *)
Lemma lexprodtripel_strictorder :
  (strictorder A gtA)->
  (strictorder B gtB)->
  (strictorder C gtC)->
  (strictorder (sigStripel A B C) (lexordtripel A B C gtA gtB gtC)).

Proof.
Intros strictorderA strictorderB strictorderC.
Unfold strictorder.
Split.

(*irreflexive*)
Unfold irreflexive.
Intro.
Unfold not.
Intro.
Inversion H.

(* Case left_lex *)
Unfold strictorder in strictorderA.
Elim strictorderA.
Unfold irreflexive.
Intros.
Apply H5 with
```

```
x:=a.
Exact H2.

(*Case middle_lex*)
Unfold strictorder in strictorderB.
Elim strictorderB.
Unfold irreflexive.
Intros.
Apply H4 with
x:= b.
Exact H2.

(*Case right_lex*)
Unfold strictorder in strictorderC.
Elim strictorderC.
Unfold irreflexive.
Intros.
Apply H3 with
x:= c.
Exact H2.

(* transitive *)
Unfold transitive.
Intros.

(* Case distinction is made by Inversion *)
Inversion H;
Inversion H0.

(* Case left_lex, a1>a2 /\ a2>a3 *)
Apply left_lex_tripel.
Rewrite <- H3 in H5.
Cut a0=a'.
Intro.
Rewrite <- H7 in H1.
Unfold strictorder in strictorderA.
Elim strictorderA.
Unfold transitive.
Intros.
Apply H9 with
x:=a
y:=a0
z:=a'0.
Exact H1.
Exact H4.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a' b' c')).
Rewrite H5.
Reflexivity.

(* Case left_lex, a1>a2 /\ (a2=a3 /\ b2>b3)*)
Apply left_lex_tripel.
Rewrite <- H3 in H5.
Cut a0=a'.
```

```
Intro.
Rewrite <- H7 in H1.
Exact H1.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a' b' c')).
Rewrite H5.
Reflexivity.

(* Case left_lex, a1>a2 /\ (a2=a3 /\b2= b3 /\ c2>c3)*)
Apply left_lex_tripel.
Rewrite <- H3 in H5.
Cut a0=a'.
Intro.
Rewrite <- H7 in H1.
Exact H1.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a' b' c')).
Rewrite H5.
Reflexivity.

(* Case left_lex, (a1=a2 /\ b1>b2) /\ a2>a3 *)
Apply left_lex_tripel.
Rewrite <- H3 in H5.
Cut a0=a.
Intro.
Rewrite H7 in H4.
Exact H4.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a b' c')).
Rewrite H5.
Reflexivity.

(* Case middle_lex, (a1=a2 /\ b1>b2) /\ (a2=a3 /\ b2>b3)*)
Rewrite <- H3 in H5.
Cut a0=a.
Intros.
Rewrite H7.
Apply middle_lex_tripel.
Cut b'=b0.
Intro.
Unfold strictorder in strictorderB.
Elim strictorderB.
Unfold transitive.
Intros.
Rewrite <- H8 in H4.
Apply H10 with
x:= b
y:= b'
z:= b'0.
Exact H1.
Exact H4.
Apply sym_eq.
Change (projS2tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS2tripel A B C (existStripel A B C a b' c')).
```

```
Rewrite H5.
Reflexivity.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a b' c')).
Rewrite H5.
Reflexivity.

(* Case middle_lex,(a1=a2 /\b1>b2) /\ (a2=a3 /\b2=b3 /\ c2>c3) *)
Rewrite <- H3 in H5.
Cut a0=a.
Intros.
Rewrite H7.
Apply middle_lex_tripel.
Cut b'=b0.
Intro.
Rewrite H8 in H1.
Exact H1.
Apply sym_eq.
Change (projS2tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS2tripel A B C (existStripel A B C a b' c')).
Rewrite H5.
Reflexivity.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a b' c')).
Rewrite H5.
Reflexivity.

(* Case left_lex, (a1=a2 /\ b1=b2 /\ c1>c2) /\ a2>a3 *)
Apply left_lex_tripel.
Rewrite <- H3 in H5.
Cut a0=a.
Intro.
Rewrite H7 in H4.
Exact H4.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a b' c')).
Rewrite H5.
Reflexivity.

(* Case middle_lex,(a1=a2 /\b1=b2 /\ c1>c2) /\ (a2=a3 /\ b2>b3) *)
Rewrite <- H3 in H5.
Cut a0=a.
Intros.
Rewrite H7.
Apply middle_lex_tripel.
Cut b=b0.
Intro.
Rewrite <- H8 in H4.
Exact H4.
Apply sym_eq.
Change (projS2tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS2tripel A B C (existStripel A B C a b c')).
Rewrite H5.
Reflexivity.
```

```
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a b' c')).
Rewrite H5.
Reflexivity.

(* Case right_lex,(a1=a2 /\ b1=b2 /\ c1>c2) /\ (a2=a3 /\ b2=b3 /\ c2>c3) *)
Rewrite <- H3 in H5.
Cut a0=a.
Intro.
Rewrite H7.
Cut b0=b.
Intro.
Rewrite H8.
Apply right_lex_tripel.
Cut c'=c0.
Intro.
Unfold strictorder in strictorderC.
Elim strictorderC.
Unfold transitive.
Intros.
Rewrite H9 in H1.
Apply H11 with
x:=c
y:=c0
z:=c'0.
Exact H1.
Exact H4.
Apply sym_eq.
Change (projS3tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS3tripel A B C (existStripel A B C a b c')).
Rewrite H5.
Reflexivity.
Change (projS2tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS2tripel A B C (existStripel A B C a b c')).
Rewrite H5.
Reflexivity.
Change (projS1tripel A B C (existStripel A B C a0 b0 c0)) =
       (projS1tripel A B C (existStripel A B C a b c')).
Rewrite H5.
Reflexivity.
Qed.

End Strictorder.

Section Wellfounded.

Variable A : Set .
Variable B : Set .
Variable C : Set .
Variable gtA : A -> A -> Prop.
Variable gtB : B -> B -> Prop .
Variable gtC : C -> C -> Prop.

(* theorem 5.16 *)
```

```
Lemma lexprodtripel_wellfounded:
 (wellfounded A gtA) ->
 (wellfounded B gtB) ->
 (wellfounded C gtC) ->
 (wellfounded (sigStripel A B C) (lexordtripel A B C gtA gtB gtC)) .

Proof.
Unfold wellfounded.
Intros WFA WFB WFC.
Induction a.
Rename a into x.

(* lemma 5.17 *)
Intro a.

(* applying well-founded induction on a *)
Apply wf_ind with
P:= [a:A](b:B)(c:C)(Access (sigStripel A B C)
                   (lexordtripel A B C gtA gtB gtC) (existStripel A B C a b c))
gtA:= gtA.
Unfold wellfounded.
Apply WFA.

(* lemma 5.18 *)
Intro a0.
Intro IHA.
Intro b.

(* applying well-founded induction on b *)
Apply wf_ind with
P:= [b:B](c:C)(Access (sigStripel A B C)
               (lexordtripel A B C gtA gtB gtC) (existStripel A B C a0 b c))
gtA:= gtB.
Unfold wellfounded.
Apply WFB.

(* lemma 5.19 *)
Intro b0.
Intro IHB.
Intro c.

(* applying well-founded induction on c *)
Apply wf_ind with
P:= [c:C](Access (sigStripel A B C)
         (lexordtripel A B C gtA gtB gtC) (existStripel A B C a0 b0 c))
gtA:= gtC.
Unfold wellfounded.
Apply WFC.

(* lemma 5.20 *)
Intro c0.
Intro IHC.

(* using the definition of accessible *)
```

```
Apply Access_intro.
Induction y.
Intros k l m.
Intro H.

(* Case distinction is made by inversion *)
Inversion H.

(* Case a0>k *)
Apply IHA with y:= k.
Exact H1.
Rewrite <- H4.

(* Case a0=k /\ b0 >l *)
Apply IHB with y:= l.
Exact H1.
Rewrite <- H4.
Rewrite <- H5.

(* Case a0=k /\ b0=l /\ c0 > m *)
Apply IHC with y:= m.
Exact H1.
Qed.

End Wellfounded.
```

Appendix G: Coq file Multisets.v

```
(* This file is about multisets and the multiset reduction *)
(* It contains three sections: *)
(* definitions: the multiset is defined, as well as equality and remove *)
(* ordering_multiset: defines a reduction on multisets *)
(* wellfounded_multiset: proves multiset reduction is wellfounded *)

Load Lex3 .

(* *************************  SECTION DEFINITIONS  ********************** *)
Section definitions .

Require PolyList.
(* finite multisets over A are represented as lists over A *)
(* with equality using the multiplicity of an element in a multiset *)

(* the elements of the multiset are in A *)
Variable A : Set .

(* equality on A is decidable *)
Hypothesis Aeq_dec : (x,y:A){eq A x y} + {~(eq A x y)} .

(* definition of the multiplicity of an element in a multiset *)
(* here it is needed that equality on A is decidable *)
Fixpoint multiplicity [a:A;m:(list A)] : nat :=
Cases m of
  nil => O
| (cons b n) => Cases (Aeq_dec a b) of
                  (left _ )  => (S (multiplicity a n))
                | (right _ ) => (multiplicity a n)
                  end
end.

(* equality on multisets over A *)
Definition eq_M : (list A) -> (list A) -> Prop :=
  [m,n:(list A)] (a:A) (multiplicity a m) = (multiplicity a n) .

(* equality is reflexive *)
Lemma eq_M_refl :
  (m:(list A))
  (eq_M m m) .

Proof .
Intro m.
```

```
Unfold eq_M .
Reflexivity .
Qed .

(* equality is symmetric *)
Lemma eq_M_sym :
  (m,n:(list A))
  (eq_M m n) -> (eq_M n m).
Proof.
Unfold eq_M .
Intros .
Apply sym_eq .
Apply H .
Qed .

(* equality is transitive *)
Lemma eq_M_trans :
  (m,n,k:(list A))
  (eq_M m n) -> (eq_M n k) -> (eq_M m k) .

Proof .
Intros m n k.
Unfold eq_M .
Intros H I .
Intro a .
Apply trans_eq with
  y := (multiplicity a n) .
Apply H .
Apply I .
Qed .

(* remove an element a from a multiset m represented as a list *)
Fixpoint remove_M [a:A;m:(list A)]: (list A) :=
Cases m of
  nil => (nil A)
| (cons b n) => Cases (Aeq_dec a b) of
                  (left _ ) => n
                | (right _ ) => (cons b (remove_M a n))
                  end
end .

End definitions .

(* ***************************  SECTION ORDERING ************************** *)
Section ordering_multiset .

Variable A : Set .
Variable gtA : A -> A -> Prop .  (* the > ordering on A *)
Hypothesis Aeq_dec : (x,y:A){eq A x y} + {~(eq A x y)} .

(* element greater than a multiset *)
Definition gt_EM : A -> (list A) -> Prop :=
  [a:A][m:(list A)](b:A)(In b m) -> (gtA a b) .
```

```
(* multiset reduction based on gtA *)
Inductive gt_M : (list A) -> (list A) -> Prop :=
  gtm:    (a:A)(n,m,m',k: (list A))
          (gt_EM a k) ->
          (eq_M A Aeq_dec m (cons a m')) ->
          (eq_M A Aeq_dec n (app k m')) ->
          (gt_M m n).

End ordering_multiset.

(* ************************** SECTION WELLFOUNDED ******************** *)
Section wellfounded_multiset.

Variable A : Set .
Variable gtA : A -> A -> Prop .  (* the > ordering on A *)
Hypothesis Aeq_dec : (x,y:A){eq A x y} + {~(eq A x y)} .


(* Hypotheses due to interaction between eq_M, Access, cons, remove_M, In and
   app *)

(* If x and y are equal and x is accessible, then y is accessible too. *)
Hypothesis acc_eq :
  (x,y:(list A))
  (eq_M A Aeq_dec x y) ->
  (Access (list A) (gt_M A gtA Aeq_dec) x) ->
  (Access (list A) (gt_M A gtA Aeq_dec) y).

(* If cons a x and cons a y are equal, then x and y are equal *)
Hypothesis cons_eq :
  (x,y:(list A))
  (a:A)
  (eq_M A Aeq_dec (cons a x) (cons a y)) ->
  (eq_M A Aeq_dec x y).

(* If x and y are equal, then cons a x and cons a y are equal *)
Hypothesis eq_cons :
  (x,y:(list A))(a:A)
  (eq_M A Aeq_dec x y ) ->
  (eq_M A Aeq_dec (cons a x) (cons a y)).

(* If a is not equal to b and cons a x and cons b y are equal,
   then y is equal to (remove b from x and add a) *)
Hypothesis cons_cons_rem :
 (x,y:(list A))(a,b:A)
  (not (eq A a b)) ->
  (eq_M A Aeq_dec (cons a x) (cons b y)) ->
  (eq_M A Aeq_dec y (cons a (remove_M A Aeq_dec b x))).

(* If  a is not equal to b and cons a x and cons b y are equal,
   then b is in x *)
Hypothesis cons_elt :
  (x,y:(list A))(a,b:A)
  (not (eq A a b)) ->
```

```
  (eq_M A Aeq_dec (cons a x) (cons b y)) ->
  (In b x).

(* Adding a and b to the union of x and y is equal to adding by to the union of
   x and a added to y *)
Hypothesis cons_app :
  (x,y:(list A))(a,b:A)
  (eq_M A Aeq_dec (cons a (cons b (app x y)))
                  (cons b (app x (cons a y )))).

(* If x and y are equal, then the union of z and x is equal to the union of
   z and y *)
Hypothesis eq_app :
  (x,y,z:(list A))
  (eq_M A Aeq_dec x y) ->
  (eq_M A Aeq_dec (app z x) (app z y)).

(* x is equal to adding a to remove a from x *)
Hypothesis cons_rem :
  (x:(list A))(a:A)
  (eq_M A Aeq_dec x (cons a (remove_M A Aeq_dec a x))).

(* Hypothesis due to interaction bewteen equality and multiset reduction *)

(* If x and x' and y and y' are equal and x > y, then x' > y' *)
Hypothesis eq_gt :
  (x,x',y,y':(list A))
  (eq_M A Aeq_dec x x') ->
  (eq_M A Aeq_dec y y') ->
  (gt_M A gtA Aeq_dec x y) ->
  (gt_M A gtA Aeq_dec x' y').

(* If a is in x, then is x greater then ( remove a from x) *)
Hypothesis rem_elt_gt :
  (x:(list A))(a:A)
  (In a x) ->
  (gt_M A gtA Aeq_dec x (remove_M A Aeq_dec a x)).

(* Lemma 6.14 *)
Lemma lemma_three :
  (a:A)
  (Access A gtA a)
->
  ((b:A)
  (gtA a b) ->
  (p: (list A)) (Access (list A) (gt_M A gtA Aeq_dec) p) ->
  (Access (list A) (gt_M A gtA Aeq_dec) (cons b p)))
->
  ((m:(list A))
  (Access (list A) (gt_M A gtA Aeq_dec) m)
->
  ((n:(list A))
  (gt_M A gtA Aeq_dec m n) ->
  (Access (list A) (gt_M A gtA Aeq_dec) (cons a n)))
```

```
->
  (Access (list A) (gt_M A gtA Aeq_dec) (cons a m))

  ).

Proof.
Intros.

(* Using the definition of accessible *)
Apply Access_intro.

(* Introducing n *)
Intro n.
Intros.

(* Case distiction of multiset reduction is made by Inversion *)
Inversion H3.

(* There is only one case : gtm *)

(* Case distinction between a = a0 and a not a0 is made by Elim *)
Elim Aeq_dec with
  x := a
  y := a0.

(* Case I: a =a0 *)
Intro.
Apply acc_eq with
  x := (app k m').
Apply eq_M_sym.
Exact H6.

Clear H6.

(* Induction on k *)
Induction k.

(* k = {{}} *)
Simpl.
Rewrite <- a1 in H5.
Apply acc_eq with
  x := m.
Apply cons_eq with
  a := a.
Exact H5.
Exact H1.

(* k > {{}} *)
Apply acc_eq with
  x := (cons a2 (app k m')).
Simpl.
Apply eq_M_refl.
Apply H0 .
Rewrite <- a1 in H4.
```

```
Unfold gt_EM in H4.
Apply H4.
Simpl.
Left.
Reflexivity.

Apply Hreck.
Unfold gt_EM.
Unfold gt_EM in H4.
Intros.
Apply H4.
Simpl.
Right.
Exact H6.

(* Case II : a not a0 *)
Intro.
Apply acc_eq with
  x := (app k m').
Apply eq_M_sym.
Exact H6.
Clear H6.

(*Induction on k*)
Induction k.

(* k = {{}} *)
Change (Access (list A) (gt_M A gtA Aeq_dec) m').
Apply acc_eq with
  x := (cons a (remove_M A Aeq_dec a0 m)).
Apply eq_M_sym.
Apply cons_cons_rem.
Exact b.
Exact H5.

Apply H2.
Apply rem_elt_gt.
Apply cons_elt with
  a := a
  y := m'.
Exact b.
Exact H5.

(* k > {{}} *)
Apply acc_eq with
  x := (cons a (app (cons a1 k) (remove_M A Aeq_dec a0 m))).
Apply eq_M_trans with
  n := (app (cons a1 k) (cons a (remove_M A Aeq_dec a0 m))).
Simpl.
Apply cons_app.

Apply eq_app.
Apply eq_M_sym.
Apply cons_cons_rem.
```

```
Exact b.
Exact H5.

Apply H2.
Apply gtm_a with
  a := a0
  n := (app (cons a1 k) (remove_M A Aeq_dec a0 m))
  m := m
  m' := (remove_M A Aeq_dec a0 m)
  k := (cons a1 k).
Exact H4.
Apply cons_rem.
Apply eq_M_refl.
Qed.

(* lemma 6.13 *)
Lemma lemma_two :
    (x:A)
    (Access A gtA x)

    ->((y:A)
        (gtA x y)
        ->(m:(list A))
           (Access (list A) (gt_M A gtA Aeq_dec) m)
           ->(Access (list A) (gt_M A gtA Aeq_dec) (cons y m))
      )
    ->(m:(list A))
       (Access (list A) (gt_M A gtA Aeq_dec) m)
       ->(Access (list A) (gt_M A gtA Aeq_dec) (cons x m)).

Proof.
Intros.

(* Applying well-founded part induction on M *)
Apply wf_part_ind with
  P := [m:(list A)](Access (list A) (gt_M A gtA Aeq_dec) (cons x m))
  gtA := (gt_M A gtA Aeq_dec).
Intros.
Elim H1.
Intros.

(* Using lemma 3 *)
Apply lemma_three.

(* Showing well-founded part induction was indeed allowed *)
Exact H.
Exact H0.
Exact H2.
Exact H3.
Exact H1.
Qed.

(* lemma 6.12 *)
Lemma lemma_one :
```

```
(a:A)(Access A gtA a) ->
(m:(list A))(Access (list A) (gt_M A gtA Aeq_dec) m) ->
(Access (list A) (gt_M A gtA Aeq_dec) (cons a m)).

Proof.
Intro a.
Intro WFa.

(* Applying well-founded part induction on a *)
Apply wf_part_ind with
  P := [a:A]
       (m:(list A))
       (Access (list A) (gt_M A gtA Aeq_dec) m)
      ->(Access (list A) (gt_M A gtA Aeq_dec) (cons a m))
  gtA := gtA.

(* Using lemma 2*)
Apply lemma_two.

(* Showing well-founded part induction was allowed *)
Apply WFa.
Qed.

Hypothesis empty_is_wellfounded :
  (Access (list A) (gt_M A gtA Aeq_dec) (nil A) ).

(* lemma 6.11 *)
Lemma elements_inW_implies_multiset_inW :
  (m:(list A))
  ((a:A)(In a m) -> (Access A gtA a))
  ->
  (Access (list A)(gt_M A gtA Aeq_dec) m) .

Proof .
Intro m .
Intro H .

(* Induction to the size of M *)
Induction m .

(* M = {{}} *)
Apply empty_is_wellfounded .

(* M > {{}} *)
Apply lemma_one.
Apply H.
Simpl.
Left.
Reflexivity.
Apply Hrecm.
Intros.
Apply H.
Simpl.
Right.
```

```
Exact H0.
Qed.

(* theorem 6.10 *)
Theorem multisets_wellfounded:
  (wellfounded A gtA)
  ->
  (wellfounded (list A) (gt_M A gtA Aeq_dec)).
Proof.
Unfold wellfounded.
Intro H.
Intro m.

(* induction to the size of M *)
Induction m.

(* M = {{}} *)
Apply empty_is_wellfounded.

(* M > {{}} *)
Apply elements_inW_implies_multiset_inW  .
Intro.
Intro.
Apply H with a:=a0.
Qed.

End wellfounded_multiset.
```

Appendix H: Coq file RPO.v

```
(* This file is about rpo and well-foundedness of rpo *)
(* It contains the following two sections: *)

(* rpo: RPO is defined with multiset reduction, one lemma is proved *)
(* rpo_wellfounded : Proved rpo is wellfounded under certain hypotheses *)

Load Multisets.
Load Terms.

(* ***************************  SECTION RPO ******************************** *)
Section rpo.

(* equality on terms is decidable *)
Hypothesis termeq_dec : (s,t:term){s=t}+{~s=t} .

(* definition 7.2: definition of rpo *)
Inductive rpo : term -> term  -> Prop :=
rpo_one :
  (f,g:nat)(ss,ts:termlist)
  (gt f g)
  ->
  ((ti:term) (in_termlist ti ts) -> (rpo (buildterm  f ss) ti))
  ->
  (rpo (buildterm  f ss) (buildterm  g ts))
|
rpo_two :
  (f:nat)(si:term)(ss,ts,us:termlist)
  (in_termlist si ss)
  ->
  ((ui:term)(in_termlist ui us) -> (rpo si ui))
  ->
  (eq_M term termeq_dec
    (termlist_to_list ts)
    (app (termlist_to_list us)
         (remove_M term termeq_dec si (termlist_to_list ss))
    )
  )
  ->
  ((ti:term) (in_termlist ti ts) -> (rpo (buildterm  f ss) ti))
  ->
  (rpo (buildterm f ss) (buildterm f ts))
|
rpo_three_a :
```

```
  (f:nat)(si,t:term )(ss:termlist)
  (rpo si t)
  ->
  (in_termlist  si ss)
  ->
  (rpo  (buildterm f ss) t)
|
rpo_three_b :
  (f:nat)(si,t:term)(ss:termlist)
  (eq term si t)
  ->
  (in_termlist si ss)
  ->
  (rpo (buildterm f ss) t).

(* ************************** lemma subtermproperty ******************** *)

(* lemma 7.4 *)
Lemma subtermproperty: (f:nat)(ss:termlist)
  ((si:term) (in_termlist si ss) -> (rpo (buildterm f ss) si)).

Proof.
Intros f ss si.
Intro.
Apply rpo_three_b with
si:=si.
Reflexivity.
Exact H.
Qed.

End rpo.

(* ****************************** SECTION RPO WELLFOUNDED *************** *)

Section rpo_wellfounded .

Hypothesis termeq_dec : (s,t:term ){s=t}+{~s=t}.

(* The statement and proof of KeyLemma2 which corresponds to lemma 7.14,
   the core part of the proof of wellfoundedness of RPO, work with triples
   of the form (function symbol, list of wellfounded terms, term).
   We give the datatypes and orderings used in the triples,
   and it is shown (or assumed) that every component is wellfounded. *)

(* ****************************** the 1st component  ******************** *)
(* The first component is the set of function symbols with a wellfounded
   precedence. Here: nat with gt. *)

(* the natural numbers are wellfounded with respect to gt *)

Hypothesis wfnat:
  (wellfounded nat gt).

(* ************************************ 2nd component ***************** *)
```

```
(* the second component is the set of termlists consisting of
   wellfounded (for RPO) terms, ordered with the multiset reduction
   of RPO.
   The set of termlists with wellfounded elements is defined as a sigma type.
   For the definition of the multiset reduction, we take the
   projection in the sigma type, and a conversion from
   termlist (from the definition of terms) to (list term)
   (with list from PolyList.v) *)

(* Definition when a term is wellfounded (for RPO) *)
Definition term_wf [s:term] : Prop :=
  (Access term (rpo termeq_dec) s) .

(* definition of when a list of terms is wellfounded (for RPO) *)
Definition termlist_wf [ss:termlist] : Prop :=
  (si:term)(in_termlist si ss)->(Access term (rpo termeq_dec) si) .

(* the set of termlists satisfying termlist_wf *)
(* The name WTL comes from Wellfounded TermList *)
Definition WTL : Set :=
  (sig termlist termlist_wf) .

(* the multiset reduction of RPO on WTL *)
Definition gt_WTL : WTL -> WTL -> Prop :=
  [ss,ts:WTL]
  (gt_M term (rpo termeq_dec) termeq_dec
    (termlist_to_list (proj1_sig termlist termlist_wf ss))
    (termlist_to_list (proj1_sig termlist termlist_wf ts))
  ) .

(* ********************** WTL is wellfounded for gt_WTL ******************* *)

(* The set of termlists with wellfounded elements (WTL) has to be wellfounded
   with respect to the multiset reduction gt_WTL *)

(* Before proving this one hypothesis and two lemma's are needed *)

(* First assume that there is a list l and a term t with t in l. If all t are
   accesible with respect to RPO, then the list l is accesible with respect to
   RPO. This is stated in the following hypothesis, which corresponds to
   Lemma elements_inW_implies_multiset_inW from Multiset.v *)

(* The lemma elements_inW_implies_multiset_inW isn't applied directly in the
   proof wfWTL, because then all hypotheses of Multiset.v have to be proven.
   Be sure access_for_lists is equal to elements_inW_implies_multiset_inW *)

Hypothesis access_for_lists :
  (l:(list term))
  ((t:term)(In t l) -> (Access term (rpo termeq_dec) t))
  ->
  (Access (list term) (gt_M term (rpo termeq_dec) termeq_dec) l) .


(* Conversion between accesibility of a list of terms (l) and accessibility
```

```
   of the first projection of a WTL (x) is stated in the lemma below. *)
(* This lemma states that if is given that l is accessible with respect to
   the multisetextension of RPO and l is equal to the first projection of x,
   where this termlist converted to a normal list, then x is accessible with
   respect to gt_WTL *)

Lemma access_between_list_and_WTL :
  (l: (list term))
  (Access (list term) (gt_M term (rpo termeq_dec) termeq_dec)l )
  ->
  ((x:WTL)
  (l = (termlist_to_list (proj1_sig termlist termlist_wf x)))
  ->
  (Access WTL gt_WTL x)) .

Proof .
Intro l.
Intro H .
Induction H .
Intros .
Apply Access_intro .
Intros .
Apply H0 with
  y := (termlist_to_list (proj1_sig termlist termlist_wf y)) ; Auto .
Rewrite H1 .
Exact H2 .
Qed .

(* The next lemma states that when the first projection of an x of type WTL
   converted to a list of terms is accessible with respect to gt_M, then x is
   accessible with respect to gt_WTL *)

Lemma access_between_proj_and_WTL:
  (x:WTL)
  (Access (list term) (gt_M term (rpo termeq_dec) termeq_dec)
  (termlist_to_list (proj1_sig termlist termlist_wf x)))
  ->
  (Access WTL gt_WTL x) .

Proof .
Intros.
Apply access_between_list_and_WTL
  with l := (termlist_to_list (proj1_sig termlist termlist_wf x)); Auto.
Qed.

(* WTL is wellfounded with respect to gt_WTL *)
Lemma wfWTL:
  (wellfounded WTL gt_WTL).

Proof.
Unfold wellfounded.
Unfold WTL .
Intro x.
Induction x .
```

```
Apply access_between_proj_and_WTL with
  x := (exist termlist termlist_wf x p) .
Simpl .
Apply access_for_lists with
  l := (termlist_to_list x) .
Unfold termlist_wf in p .
Unfold in_termlist in p .
Exact p .
Qed .

(* ********************** 3rd component ********************************** *)
(* the 3rd component is term with ordering gt_term_length *)
(* the hypothesis states that this is wellfounded *)

(* Defines the relation between terms on the third argument of the tripel *)
Inductive gt_term_length: term->term->Prop:=
  gttermlength: (s,t:term)(gt (length_term s) (length_term t))->
                (gt_term_length s t).

(* every ground term is accessible for the ordering counting symbols *)
Hypothesis wflength: (wellfounded term gt_term_length).

(* ************************** The lemma Aux used in KeyLemma2 ************ *)

(* Lemma aux corresponds to lemma 7.15 where is proven that all ti have to be
   wellfounded *)

(* Aux is used in case 1 and case 2 of RPO. *)

(* Hypothesis about the length of a term: the length of term is greater than
   the length of one of its subterms *)

Hypothesis subtermlength :
  (t:term)(ti:term)
  (in_termlist ti (subterms t))->
  (gt (length_term t) (length_term ti)).

(* lemma 7.15 *)
Lemma Aux :
(f : nat)(g:nat)(ss : WTL)(t:term)(ts: termlist)

((buildterm g ts)=t)->
((y:(sigStripel nat WTL term))
         (lexordtripel nat WTL term gt gt_WTL gt_term_length
           (existStripel nat WTL term f ss t) y)
         ->(termlist_wf
             (proj1_sig termlist termlist_wf
               (projS2tripel nat WTL term y)))
         ->(rpo termeq_dec
             (buildterm (projS1tripel nat WTL term y)
               (proj1_sig termlist termlist_wf
                 (projS2tripel nat WTL term y)))
             (projS3tripel nat WTL term y))
         ->(term_wf (projS3tripel nat WTL term y))
```

```
)
->
((ti:term)
  (in_termlist ti ts)
  ->(rpo termeq_dec
  (buildterm f (proj1_sig termlist termlist_wf ss)) ti)
)
->
(termlist_wf ts) .

Proof.
(* Introducing natural numbers f and g, term t and termlists ss and ts. *)
Intros f g ss t ts.

(* Introducing hypotheses *)
Intro tISgts.
Intro IH1.
Intro fssRPOti.
Unfold termlist_wf.

(* Introducing ti as subterm of ts *)
Intro ti.
Intro .

(* Apply induction hypothesis for f(ss)>ti *)
Apply IH1 with
y:= (existStripel nat WTL term f ss ti).

(* Show three hypotheses hold for f(ss)>ti *)

(*I: Show that (f,(s1,...,sm),t) >LEX (f,(s1, ..., sn),ti) holds*)
Apply right_lex_tripel.
Apply gttermlength.
Apply subtermlength.
Rewrite <- tISgts.
Simpl.
Exact H.

(*II: Show that ss are in W *)
Simpl.
Induction ss.
Apply p.

(*III : Show that f (ss) >RPO ti*)
Simpl.
Apply fssRPOti with ti:=ti.
Exact H.
Qed.

(* ****************** a hypothesis concerning multisets ********************)
(* used in case rpo2 of KEYLEMMA2 *)

(* One hypothesis from the multiset module*)
Hypothesis remove_add :
```

```
  (m:(sig termlist termlist_wf))(a:term)
  (eq_M term termeq_dec
     (cons a (remove_M term termeq_dec a (termlist_to_list (proj1_sig termlist termlist_wf m))))
     (termlist_to_list (proj1_sig termlist termlist_wf m)) ).

(* ************************  KEY LEMMA 2 ********************************** *)

(* KeyLemma2 contains the core of the proof of wellfoundedness
   of RPO.
   The statement of KeyLemma2 is in terms of triples of the
   form (function symbol, list of terms that are WF for RPO, term).
   The proof of KeyLemma2 proceeds by wellfounded induction on
   such a triple, where the ordering is the
   3-fold lexicographic product (from Lex3.v)
   with the following orderings on the components:
   1. function symbol (in nat) ordered by gt
   2. list of terms that are WF for RPO (in WTL) ordered by gt_WTL
   3. term (in term) ordered by gt_term_length

   The proof of KeyLemma2 is quite hard.
   We use
   1. the lemma Aux
   2. wfnat stating that (nat,gt) is wellfounded
   3. wfWTL stating that (WTL, gt_WTL) is wellfounded
   4. wflength stating that (term, gt_term_length) is wellfounded
   5. lexprodtripel_wellfounded (from Lex3.v) stating that
      the lexicographic product of three wellfounded orderings is again
      wellfounded. *)

(* lemma 7.14 *)
Lemma KeyLemma2:
  (l: (sigStripel nat WTL term))
  (termlist_wf (proj1_sig termlist termlist_wf (projS2tripel nat WTL term l)))
  ->
  (rpo
    termeq_dec
    (buildterm (projS1tripel nat WTL term l)
               (proj1_sig termlist termlist_wf (projS2tripel nat WTL term l)))
    (projS3tripel nat WTL term l)
  )
  ->
 (term_wf (projS3tripel nat WTL term l)) .

Proof.
Intro l.

(* Applying wellfounded induction on the tripel *)
Apply wf_ind with
A := (sigStripel nat WTL term)
gtA:= (lexordtripel nat WTL    term
                    gt  gt_WTL gt_term_length)
P:= [l: (sigStripel nat WTL term)]
    (termlist_wf (proj1_sig termlist termlist_wf (projS2tripel nat WTL term l)))
    ->
```

```
    (rpo
      termeq_dec
        (buildterm (projS1tripel nat WTL term l)
               (   proj1_sig termlist termlist_wf (projS2tripel nat WTL term l)))
        (projS3tripel nat WTL term l)
     )
     ->
     (term_wf (projS3tripel nat WTL term l)) .

(* Showing that well-founded induction was indeed allowed *)
Apply lexprodtripel_wellfounded.
Apply wfnat.
Apply wfWTL.
Apply wflength.
(* Introducing f ss t *)
Induction x.
Intros f ss t.

(* Introducing hypotheses *)
Intro IH1.
Intro WFss.
Intro fssRPOt.

(* case distinction by the definition of RPO *)
Inversion fssRPOt.

(* ************************ case rpo 1******************************** *)

Unfold term_wf.

(* using the definition of accessibility, going 1 step lower in the tripel *)
Apply Access_intro.

(* Introducing u *)
Intro u.

(* Introducing hypothesis gtsRPOu *)
Intro gtsRPOu.

(* Apply induction hypothesis for g(ts) >u *)
Simpl in H2.
Apply IH1 with
  y:=(existStripel nat WTL term
        g (exist termlist termlist_wf ts (Aux f g ss t ts H2 IH1 H3)) u) .

(* Show three hypotheses hold for g(ts)>u *)

(*I: Show that (f,(s1,...,sm),t) >LEX (g,(t1, ..., tn),u) holds*)
Apply left_lex_tripel.
Exact H1.

(*II: Show that ts are in W *)
Simpl.
Apply (Aux f g ss t ts H2 IH1 H3) .
```

```
(*III : Show that g (ts) >RPO u*)
Simpl.
Exact gtsRPOu.

(* ********************** case rpo 2****************************** *)

Unfold term_wf.

(* using the definition of accessibility, going 1 step lower in the tripel *)
Apply Access_intro.

(* Introducing u *)
Intro u.

(* Introducing hypothesis gtsRPOu *)
Intro gtsRPOu.

(* Apply induction hypothesis for g(ts) >u *)
Simpl in H4.
Apply IH1 with
  y:= (existStripel nat WTL term
         f
         (exist termlist termlist_wf ts (Aux f f ss t ts H4 IH1 H5))
         u
       ).

(* Show three hypotheses hold for g(ts)>u *)

(*I: Show that (f,(s1,...,sm),t) >LEX (g,(t1, ..., tn),u) holds*)
Apply middle_lex_tripel.
Unfold gt_WTL .
Simpl .

(* Apply when a multiset gt multiset *)
Apply gtm_a with
  a := si
  n := (termlist_to_list ts)
  m := (termlist_to_list (proj1_sig termlist termlist_wf ss))
  m':= (remove_M term termeq_dec si (termlist_to_list (proj1_sig termlist termlist_wf ss)))
  k := (termlist_to_list us) .

(* Three clauses of gtm_a have to hold *)

(* 1: (gt_EM term (rpo termeq_dec) si (termlist_to_list us)) *)
Unfold gt_EM.

(* introducing ui as subterm of us *)
Intro ui.
Intro.
Apply H2 with
ui:=ui.
Unfold in_termlist.
Exact H6.
```

```
(*2:(eq_M term termeq_dec
     (termlist_to_list (proj1_sig termlist termlist_wf ss))
     (cons si
       (rem term termeq_dec si
         (termlist_to_list (proj1_sig termlist termlist_wf ss))))) *)
Simpl.
Apply eq_M_sym.
Apply remove_add with
m:=ss
a:=si.

(* 3: (eq_M term termeq_dec (termlist_to_list ts)
     (app (termlist_to_list us)
       (rem term termeq_dec si
         (termlist_to_list (proj1_sig termlist termlist_wf ss))))) *)
Exact H3.

(*II: Show that ts are in W *)
Simpl.
Apply (Aux f f ss t ts H4 IH1 H5) .

(*III : Show that g (ts) >RPO u*)
Simpl.
Exact gtsRPOu.

(* ************************ case rpo 3******************************** *)

(* case rpo 3a*)
Simpl.
Simpl in WFss.
Unfold term_wf.

(* Apply remark *)
Apply Access_inv with
x:=si .
Apply WFss.
Exact H3.
Simpl in H1.
Exact H1.

(* Case rpo 3b*)
Simpl.
Unfold term_wf.
Simpl in H1.
Rewrite <- H1.
Apply WFss.
Simpl.
Exact H3.
Qed.

(* ********************************* KEY LEMMA ************************** *)

(* The KeyLemma handles the induction case in the proof
```

```
   of the main result: if all terms in ss are wellfounded
   for RPO, then f(ss) is wellfounded for RPO
   In the proof we use KeyLemma2 which is stated in
   terms of tripels of the form
   (function symbol, list of terms that are WF for RPO, term).
*)

(* lemma 7.13 *)
Lemma KeyLemma :
  (f:nat)(ss:termlist)
  (termlist_wf ss) -> (Access term (rpo termeq_dec) (buildterm f ss)) .

Proof.
(* Introducing f ss *)
Intros  f ss.

(* Introducing wfss*)
Intro WFss.

(* Using the definition of accessible *)
Apply Access_intro.

(* Introducing y *)
Induction y.

(* Introducing g ts *)
Intros g ts.

(* Introducing fssRPOgts *)
Intro fssRPOgts.

(* Apply Keylemma 2 with the tripel (f, ss, t=g(ts)) *)
Apply KeyLemma2 with
  l:=(existStripel nat WTL term f (exist termlist termlist_wf ss WFss)
      (buildterm g ts)).

(* Show the hypotheses of Keylemma2 *)

(* ss is a wellfounded termlist *)
Simpl .
Exact WFss .

(* fss RPO gts *)
Simpl.
Exact fssRPOgts.
Qed.

(* ****************************** two small lemma's ********************** *)

(* before proving the main result two small lemma's are needed, the lemma's
   state wellfoundedness of the emptylist and wellfoundedness of the 'conslist'*)
(* the empty list is accessible *)
Lemma nilterm_wellfounded: (termlist_wf nilterm).
```

```
Proof.
Unfold termlist_wf.
Intro si.
Intro H.
Unfold in_termlist in H.
Simpl in H.
Tauto.
Qed.

(* all terms in a termlist in which all elements are wellfounded for RPO
   are wellfounded for RPO *)

Lemma wf_in_lists :
  (ti,h:term)(ts:termlist)
  (in_termlist ti (consterm h ts))
  ->
  (term_wf h)
  ->
  (termlist_wf ts)
  ->
  (term_wf ti) .

Proof.
Intros ti h ts .
Intro H .
Intro WFh .
Intro WFts .

Elim termeq_dec with
  s := ti
  t := h .

Intro I .
Rewrite I .
Exact WFh .

Intro I .
Unfold termlist_wf in WFts.
Apply WFts with si:=ti.
Unfold in_termlist.
Unfold in_termlist in H.
Elim in_inv with b:=ti a:=h l:=(termlist_to_list ts).
Intro.
Elim I.
Apply sym_eq.
Exact H0.
Intro.
Exact H0.
Exact H.
Qed.

(* ****************************** MAIN RESULT ************************** *)

(* The proof of the main result proceeds by induction
```

```
   on the structure of terms.
   We use three lemma's:
   1. nilterm_wellfounded
      stating that
      This lemma is easy.
   2. wf_in_lists stating that all terms in a termlist
      in which all terms are wellfounded for RPO are
      wellfounded for RPO
      This lemma is quite easy.
   3. KeyLemma
      stating that f(ss) is wellfounded for RPO if all
      terms in ss are wellfounded for RPO.
      This lemma is hard.
*)

(* Theorem 7.11 *)
Theorem rpo_wf:(wellfounded term (rpo termeq_dec)).

Proof.
Unfold wellfounded.

(* Theorem 7.12 *)
Intro s.

(* Applying induction on terms *)
Apply term_induction with
  P :=[t:term](Access term (rpo termeq_dec) t)
  P0:=[t0:termlist] (termlist_wf t0).

(* Show the three hypotheses for term_induction (see section 2.1) *)
(* P := (Access term (rpo termeq_dec) t) *)
(* P0:= (termlist_wf t0) *)

(* I: ((n:nat; t:termlist)(P0 t)->(P (buildterm n t))) *)
(* Introducing f ss *)
Intros f ss .

(* Introducing induction hypothesis, (P0 ss)*)
Intro IH .

(* Apply keylemma *)
Apply KeyLemma.

(* Show hypothesis of keylemma *)
Exact IH.

(* II:(P0 nilterm) *)
Apply nilterm_wellfounded.

(* III:((t:term)(P t)->(t0:termlist)(P0 t0)->(P0 (consterm t t0))) *)
(* introducing h *)
Intro h.

(* introducing induction hypothesis, (P h) *)
```

```
Intro IH1 .

(* introducing ts *)
Intro ts.

(*introducing induction hypothesis, (P0 ts) *)
Intro IH2.
Unfold termlist_wf.

(* introducing ti as subterm of ts *)
Intro ti.
Intro.

(* apply wf_in_lists *)
Apply wf_in_lists with
  ti := ti
  h  := h
  ts := ts .

(* show hypotheses of wf_in_lists *)
Exact H.
Exact IH1.
Exact IH2.
Qed.
```