Thesis for the Degree of Licentiate of Philosophy

Programming in Martin-Löf Type Theory

Unification
A non-trivial Example

Ana Bove

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, November 1999

Programming in Martin-Löf Type Theory
Unification - A non-trivial Example
Ana Bove

ISBN 91-7197-861-5

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, November 1999

Abstract

Martin-Löf's type theory is a constructive type theory originally conceived as a formal language in which to carry out constructive mathematics. However, it can also be viewed as a programming language where specifications are represented as types and programs as objects of the types. In this work, the use of type theory as a programming language is investigated. As an example, a formalisation of a unification algorithm for first-order terms is considered.

Unification can be seen as the process of finding a substitution that makes all the pairs of terms in an input list equal, if such a substitution exists. Unification algorithms are crucial in many applications, such as type checkers for different programming languages. Unification algorithms are total and recursive, but the arguments on which the recursive calls are performed satisfy no syntactic condition that guarantees termination. This fact is of great importance when working with Martin-Löf's type theory since there is no direct way of formalising such an algorithm in the theory.

The standard way of handling general recursion in Martin-Löf's type theory is by using the accessibility predicate Acc which captures the idea that an element a in A is accessible if there exists no infinite decreasing sequence starting from a. As this is a general predicate, it contains no information that can help us when formalising a particular general recursive algorithm, and then its use in the formalisation of the unification algorithm produces an unnecessarily long and complicated algorithm. On the other hand, functional programming languages like Haskell impose no restrictions on recursive programs, and then writing an algorithm like the unification algorithm is straightforward. In addition, functional programs are usually short and self-explanatory. However, there does not exist a powerful framework that allows us to reason about the correctness of Haskell-like programs.

Then, the goal of this work is to present a methodology that combines the advantages of both programming styles and that can be used for the formalisation of the unification algorithm. In this way, a short algorithm that can be proven correct by using the expressive power of constructive type theory is obtained.

The main feature of the methodology presented here is the introduction of an inductive predicate, specially defined for the unification algorithm, that can be thought of as defining the set of lists of pairs of terms for which the algorithm terminates. This predicate contains an introduction rule for each of the cases that need to be considered and provides an easy syntactic condition that guarantees termination. The information contained in this predicate simplifies the formalisation of both the algorithm and the proof of its partial correctness. In this way, both the algorithm and the proof are short, elegant and easy to follow. In addition, it is possible to define a methodology that extracts a Haskell program from the type theory formalisation of the unification algorithm that is defined by using this special-purpose predicate.

Acknowledgements

First of all, I would like to thank my supervisor Björn von Sydow for his wise help and advise. He always knows which question to ask, when is the right moment to ask it and where I should turn to find my way to its answer. He has also read and commented on different versions of this work.

Although he might not be aware of it, it was after exchanging some emails with Daniel Fridlender that I came up with the idea of how to construct the special-purpose accessibility predicate that I present in this thesis. I would like to thank him for this.

This work has benefited from discussions with several people, among others, Thorsten Altenkirch, Thierry Coquand, Peter Dybjer and Conor McBride. I would like to express my gratitude to all of them.

I am also very grateful to those who have read all or parts of previous versions of this thesis and made useful comments. I would like to thank Jörgen Gustavsson, Thomas Hallgren, Johan Jeuring, Bengt Nordström and Mary Sheeran for this.

I am thankful to all my friends, both here and there, and my family for their constant support during all this time. Special thanks go to my office mate, Andrei Sabelfeld, who still seems to be happy to see me every day in the office.

Finally, I would like to thank everybody at the Computing Science Department for providing such a friendly environment.

Contents

Chapter 1

Introduction

Martin-Löf's type theory [ML84, NPS90, CNSvS94] is a constructive type theory originally conceived as a formal language in which to carry out constructive mathematics. Following the Curry-Howard isomorphism [How80], a theorem is represented as a type and a proof of the theorem is an object of the corresponding type. Thus, a proof of a theorem is in general a function that, given proofs of the hypotheses of the theorem, computes a proof of the thesis.

However, Martin-Löf's type theory can also be seen as a programming language where specifications are represented as types and programs as objects of the types. Hence, when a specification states the existence of an object with certain properties, any program that satisfies the specification computes such an object. One can use the expressive power of type theory to reason about program correctness. This clearly is an advantage of constructive type theory over standard programming languages, and therefore the use of type theory in programming has been the object of several studies (see for example [ML82, Sza97]). In this work, we continue along this line and investigate the use of Martin-Löf's type theory as a programming language for the formalisation of a unification algorithm for first-order terms.

The unification problem for first-order terms can be stated in different ways. In our case, unification is the process of finding a substitution that makes all the pairs of terms in an input list equal, if such a substitution exists. Unification has become widely known since Robinson used it in his resolution principle [Rob65]. Unification algorithms are crucial in many applications, such as resolution and non-resolution theorem provers, computation of critical pairs for term rewriting systems, and type checkers and type inference algorithms for different programming languages. Unification algorithms for first-order terms are total and recursive. In our case study, the recursive calls are on lists of pairs of terms. Although we can define a complexity measure over lists of pairs of terms that strictly decreases in each recursive call, the recursive calls are on non-structurally smaller arguments, and then there is no easy syntactic condition that guarantees termination. This fact is of great importance when working with Martin-Löf's type theory since there is no direct way of formalising such

an algorithm in the theory.

The standard way of handling general recursion in Martin-Löf's type theory is by using the accessibility predicate Acc. This predicate captures the idea that an element a of type A is accessible (by a certain less-than relation on A) if there exists no infinite decreasing sequence starting from a. When we use this predicate to write the type theory version of a general recursive algorithm that performs the recursive calls on arguments of type A we proceed as follows. First, we add an extra argument to the formalisation of the algorithm requiring the input argument of type A to be accessible. In this way, we define the algorithm only for those inputs that are accessible. When writing the algorithm, in addition to the actual computations we need to perform, we need to provide proofs showing that the arguments on which we perform the recursive calls are smaller than the input argument. Hence, as there is no infinite decreasing sequence that starts from the initial input argument (since the argument is accessible), we guarantee that the algorithm terminates. Finally, by proving that all elements of type A are accessible we show that the algorithm is defined for all possible inputs. The problem with this formalisation is that, as the standard accessibility predicate is a general predicate, it contains no information that can help us when formalising a particular general recursive algorithm. Thus, the process we just described is not always the best way of formalising general recursive algorithms in type theory since it sometimes produces unnecessarily long and complicated algorithms.

Writing general recursive algorithms is not a problem in functional programming languages like Haskell [JHe+99] since this kind of language imposes no restrictions on recursive programs, which makes the writing of algorithms like the unification algorithm straightforward. In addition, functional programs are usually elegant, self-explanatory and shorter than their imperative versions. However, the existing frameworks that allow us to reason about the correctness of Haskell-like programs are weaker than the framework provided by type theory, and it is basically the responsibility of the programmer to only write programs that are correct.

In this work, we combine the advantages of both programming styles when writing general recursive algorithms. In this way, we can write general recursive algorithms in Martin-Löf's type theory that are short, self-explanatory and that can be proven correct by using the expressive power of type theory. As a first step in this process, we present a methodology that allows us to write a short and elegant unification algorithm in Martin-Löf's type theory.

The main feature of our methodology is the introduction of a inductive predicate, specially defined for the algorithm to be formalised, that can be thought of as defining the set of input on which the algorithm terminates and that can be used for formalising the desired algorithm in Martin-Löf's type theory. This predicate contains an introduction rule for each of the cases that need to be considered and provides an easy syntactic condition that guarantees termination.

In our case study, we call this inductive predicate UniAcc and we think of it as defining the set of lists of pairs of terms on which our unification algorithm

terminates. In other words, a list of pairs of terms lp satisfies the predicate UniAcc if our algorithm terminates on the input list lp. To define this predicate, we present a(n almost) mechanical method that, given a Haskell version of the unification algorithm, constructs an introduction rule of the predicate for each of the cases we need to consider when writing the algorithm. The method is such that, given a list of pairs of terms lp, there is one and only one introduction rule of the predicate that can be used for proving that the predicate holds for this particular list lp. In addition, the premises of this introduction rule are the necessary conditions that should be satisfied in order to ensure the termination of our unification algorithm on the input lp. Moreover, the information contained in the rule for lp is extracted from the equation of the Haskell version of the unification algorithm that defines the result of the algorithm for the list lp. Observe that, if for the input lp the unification algorithm performs a recursive call on the list lp', the unification algorithm can only terminate on the input lp if it terminates on the input lp'. Hence, a proof that the list lp' satisfies the predicate is required as a premise of the (only) introduction rule that allows us to show that the list lp satisfies our special predicate.

Once we have defined our special predicate, we proceed as follows to write the type theory version of the unification algorithm. First, we add an extra argument to the formalisation of the algorithm requiring that the input list of pairs of terms satisfies the predicate UniAcc (which means, intuitively, that the algorithm terminates on this particular input) and we define the algorithm by recursion on this extra argument. To write the algorithm we perform pattern matching over the proof that the initial list to be unified satisfies our predicate, obtaining one equation for each of the introduction rules of the predicate. As each introduction rule of the predicate corresponds to one of the cases considered in the Haskell version of the algorithm, the type theory equations of the algorithm closely follow the corresponding Haskell cases. In each of the recursive equations we should supply a proof that the new list to be unified satisfies the special predicate. Recall that, in each of the recursive equations, this information is actually one of the arguments of the proof that the initial input list satisfies the predicate UniAcc. Hence, we just need to select the corresponding argument and supply it to the recursive call. Observe that this ensures that the algorithm is defined by structural recursion on the proof that the list to be unified satisfies the predicate UniAcc. Finally, by proving that all lists of pairs of terms satisfy our special predicate, we show that the algorithm is defined for all possible inputs.

If we compare the version of the unification algorithm that uses the standard accessibility predicate (which is presented in section 4.4.1) and the version of the unification algorithm that uses our special predicate (which is presented in section 4.5.2), we see that the latter is more compact and easier to read than the former and it is exactly our special predicate that allows us to obtain this improvement. The main reasons for this improvement are the following:

- The way we define the predicate UniAcc ensures that we have one (and only one) introduction rule for each of the cases we need to consider.

In this way, by doing pattern matching over the proof that the input list satisfies the predicate we obtain, at once, all the different cases we need to consider. On the other hand, if we use the standard accessibility predicate Acc for defining the unification algorithm we need to do several case analyses to obtain the different cases we need to consider.

- Our predicate UniAcc provides an easy syntactic condition that guarantees termination if we define the algorithm by recursion on the proof that the argument to be unified satisfies the predicate.
- The proofs that the lists on which we perform the recursive calls are smaller than the original list are essential to guarantee the termination of the type theory version of the unification algorithm that uses the standard accessibility predicate to handle the recursive calls. However, these proofs add a considerable amount of code to the algorithm and also distract our attention from the actual unification process since they do not have any computational content.

  As these proofs are not needed in the type theory version of the unification algorithm that uses the predicate UniAcc to handle the recursive calls (since this predicate provides an easy syntactic condition that guarantees termination), the resulting algorithm gets considerably shorter and clearer.

  Actually, our special predicate allows us to move these long proofs from the real unification process to the proof that the predicate UniAcc holds for all possible inputs. We show that our special predicate holds for all possible inputs (and hence, that our unification algorithm is total) in a function that is completely separate from the actual unification process. Unfortunately, we did not come up with a nice proof that our predicate holds for all possible inputs.

In addition to the type theory code of the unification algorithm and the proof that it is defined for all possible inputs, we also prove several lemmas that show that the algorithm is partially correct. Here, we can see that these proofs also benefit from the definition of our special predicate. Mainly for the reasons pointed out above, these proofs are short and easy to follow.

Finally, we integrate the actual unification process and its partial correctness into the complete specification of our unification algorithm. Once more, we use our special predicate in this integrated approach for the unification algorithm in Martin-Löf's type theory and benefitted from its definition in the same way as before.

Both the formalisation of the unification algorithm and all the properties we present in this work have been machine-checked in ALF [Mag92, AGNvS94, MN94], which is an interactive proof assistant for Martin-Löf's type theory extended with pattern matching [Coq92]. Although ALF does not support program extraction, we discuss a methodology that allows us to extract a Haskell program from the formalisation of the unification algorithm that uses our special predicate to handle the recursive calls. The Haskell version of the unification algorithm that results from applying our program-extraction methodology (which

is presented in section 4.6) is very similar to the Haskell version of the algorithm that we used for constructing the predicate UniAcc. Actually, we can think of the process that given a Haskell version of the unification algorithm constructs its type theory version, and the process that extracts a Haskell program from our type theory version of the algorithm as inverses of each other.

We end this section by summarising the advantages of the methodology we present in this work to construct our type theory version of the unification algorithm.

- The way we define our special predicate ensures that we have one introduction rule for each of the cases we need to consider when writing the algorithm.
- Our special predicate provides an easy syntactic condition that guarantees the termination of the algorithm.
- Our special predicate allows us to separate the actual unification process and its partial correctness from the total correctness of it.
- These three points allow us to obtain a type theory version of the unification algorithm that is short and elegant.
- For the reasons pointed out above, our special predicate also simplifies all the proofs we present in this work. Each of the proofs we present here is short and elegant.
- The methodology we present here allows us to extract a Haskell algorithm from our type theory version of the unification algorithm.
- Finally, we believe that the methodology we use for the unification algorithm can be used for writing other total and general recursive algorithms in type theory. In this way, we think that this methodology gives a step towards closing the existing gap between programming in a Haskell-like programming language and programming in Martin-Löf's type theory. However, the generalisation of this methodology to all total and general recursive algorithms remains to be studied.

This paper is intended for readers who have some basic knowledge of Martin-Löf's type theory. In what follows, sometimes we use "type theory" as an abbreviation for "Martin-Löf's type theory".

1.1 Overview of the Thesis

This thesis is organised as follows:

In chapter 2, we give a brief introduction to Martin-Löf's type theory and its interactive proof assistant ALF, and we present some general set formers and constructors in Martin-Löf's type theory together with some of their operators and properties.

In chapter 3, we present the formalisation of a small and general recursive algorithm in Martin-Löf's type theory: division-by-two over natural numbers. By using this very simple example, we illustrate the methodology we introduce here for writing general recursive algorithms in type theory. In addition, we show the advantages of this methodology over the standard way of defining general recursive algorithms in type theory, which is by using the accessibility predicate Acc.

In chapter 4, we introduce the Haskell version of the unification algorithm that we consider and we give an informal explanation of its termination. After introducing the necessary definitions in ALF, we describe how we can write the unification algorithm in type theory by using the standard accessibility predicate, and we show why the use of this predicate to write the unification algorithm in type theory is not a good solution in our case. Then, we present a special predicate for the unification algorithm and we show how we can use this predicate to write the unification algorithm in Martin-Löf's type theory, obtaining a short and self-explanatory algorithm. Finally, we discuss a methodology that allows us to extract a Haskell program from the formalisation of the unification algorithm that uses our special predicate to handle the recursive calls.

In chapter 5, we introduce a few more definitions and some properties of substitutions which will be used in the following two chapters to prove the partial correctness of the algorithm Unify and to present the integrated approach to the unification algorithm.

In chapter 6, we present the partial correctness of our formalisation of the unification algorithm. That is, we prove that the unification algorithm returns the value error only if there exists no substitution that unifies the input list of pairs of terms; otherwise it returns an idempotent substitution that is a most general unifier of the input list of pairs of terms and whose variables are included in the set of variables in the input list.

In chapter 7, we present the internal or integrated approach to the unification algorithm. In other words, we integrate the actual unification process and its partial correctness into a complete specification of the unification algorithm, and we prove that there is an object that satisfies this specification.

In chapter 8, we present some conclusions, related work and future work.

In appendix A, we explain the ALF proofs of the inequalities over lists of pairs of terms presented in section 4.2 to justify the termination of the unification algorithm.

In appendix B, we discuss the proof that shows that all lists of pairs of terms satisfy the special predicate UniAcc.

Finally, appendix C contains the complete ALF code for the formalisation of the unification algorithm in Martin-Löf's type theory.

Chapter 2

Introduction to Martin-Löf's Type Theory and ALF

Here, we first give a brief introduction to Martin-Löf's type theory and its interactive proof assistant ALF, and then we present some general set formers and constructors in Martin-Löf's type theory, together with some of their operators and properties.

2.1 Brief Introduction to Martin-Löf's Type Theory

Although this paper is intended mainly for those who already have some knowledge of type theory, and in particular of Martin-Löf's type theory, we present in this section a brief introduction to this theory to make the following sections more readable. For a more complete introduction to the subject, the reader can refer to [ML84, NPS90, CNSvS94].

Martin-Löf's type theory has a basic type and two type formers. The basic type is the type of sets. For each set S, the elements of S form a type. Given a type α and a family β of types over α, we can construct the function type from α to β. We write $a \in \alpha$ for "a is an object of type α".

Sets, elements of sets and functions are explained as follows:

Sets: Sets are inductively defined. In other words, a set is determined by the rules that construct its elements, that is, the set's constructors. We write **Set** to refer to the type of sets.

Elements of Sets: For each set S, the elements of S form a type called $\mathbf{El}(S)$. However, for simplicity, if a is an element in the set S, we say that a has type

S, and thus we simply write $a \in S$ instead of $a \in \mathbf{El}(S)$.

Dependent (and non-dependent) Functions: A dependent function is a function in which the type of the output depends on the value of the input. To form the type of a dependent function, we first need a type α as domain and then a family of types over α. If β is a family of types over α, then to every object a of type α there is a corresponding type $\beta(a)$.

Given a type α and a family β of types over α, we write $(x \in \alpha)\beta(x)$ for the type of dependent functions from α to β. If f is a function of type $(x \in \alpha)\beta(x)$ then, when applying f to an object a of type α we obtain an object of type $\beta(a)$. We write $f(a)$ for such an application.

A (non-dependent) function is considered a special case of a dependent function, where the type β does not depend on a value of type α. When this is the case, we may write $(\alpha)\beta$ for the function type from α to β.

Predicates and relations are seen in type theory as functions yielding propositions as output. As well as sets, propositions are inductively defined. So, a proposition is determined by the rules that construct its proofs. To prove a proposition P, we have to construct an object of type P. In other words, a proposition is true if we can build an object of type P and it is false if the type P is not inhabited. We write **Prop** to refer to the type of propositions. However, the way propositions are introduced allows us to identify propositions and sets, and then we usually write **Set** instead of **Prop**.

2.2 Brief Introduction to ALF

ALF (Another Logical Framework) is an interactive proof assistant for Martin-Löf's type theory extended with pattern matching [Coq92]. In Martin-Löf's type theory, theorems are identified with types and a proof is an object of the type, generally a function mapping proofs of the hypotheses into proofs of its thesis. ALF ensures that the constructed objects are well-formed and well-typed. Since proofs are objects, checking well-typing of objects amounts to checking correctness of proofs. For more information about ALF see [Mag92, AGNvS94, MN94].

A set former, or in general, any inductive definition is introduced as a constant S of type $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\mathbf{Set}$, for $\alpha_1, \ldots, \alpha_n$ types. For each set former, we have to introduce the constructors associated with the set which construct the elements of $S(a_1, \ldots, a_n)$, for $a_1 \in \alpha_1, \ldots, a_n \in \alpha_n$.

Abstractions are written in ALF as $[x_1, \ldots, x_n]\, e$ and theorems are introduced as dependent types of the form $(x_1 \in \alpha_1; \ldots; x_n \in \alpha_n)\beta(x_1, \ldots, x_n)$. If the name of a variable is not important, one can simply write (α) instead of $(x \in \alpha)$, both in the introduction of inductive definitions and in the declaration of (dependent) functions. Whenever $(x_1 \in \alpha;\ x_2 \in \alpha; \ldots;\ x_n \in \alpha)$ occurs, ALF displays $(x_1,\ x_2, \ldots,\ x_n \in \alpha)$ instead.

A function can be defined by performing pattern matching over one (or more) of its arguments. The various cases in the pattern matching are exhaustive and mutually disjoint. Moreover, they are computed by ALF according to the definition of the set to which the selected argument belongs. In general, theorems are proven by primitive recursion on one of its arguments. Unfortunately, ALF does not check well-foundedness when working with recursive proofs. However, for the proofs we present in this paper termination is guaranteed because we always apply the recursion on a structurally smaller argument. When proving a theorem by recursion on an argument a of type A, we first perform pattern matching over the argument a to obtain all the possible cases for a. Then, to each of the recursive calls we supply the corresponding proper sub-pattern of the proof a, which should be of type A. In this way, checking well-foundedness in our proofs is easy – even if rather tedious – to perform manually.

Sometimes, it is useful to define a function by doing case analysis on an element a of type A. For this, we can use ALF's **case** expression. The result of considering cases on $a \in A$ is similar to the result of performing pattern matching over a. The difference is that, when we do case analysis on a, a does not need to be an argument of the function we want to define but any proof of A. Hence, we can use any other previously defined function to construct the proof a of A. Once again, the various cases in the case analysis are exhaustive, mutually disjoint and computed by ALF according to the definition of the set A.

The following example shows how we can return the value zero or one depending on whether or not the natural numbers n and m are equal.

```
ex1 ∈ (n, m ∈ N) N
    ex1(n, m) ≡ case N_dec(n, m) ∈ Dec(=(n, m)) of
                    yes(h) ⇒ 0
                    no(h) ⇒ 1
                end
```

Here, we perform case analysis on the proof that the equality of n and m is decidable and we obtain two cases: either the numbers are equal or not. In the first case, both numbers are equal (and h is a proof of that) and we return the value zero. In the second case, the numbers are not equal (and h is a proof of that) and we return the value one. Although it was not necessary in our very simple example, we can use the argument h as part of the resulting value in each of the two cases.

In particular, if we do case analysis on a proof a of absurdity, that is, we have $a \in \bot$, we do not obtain any case to study since there does not exist any proof of absurdity. In this way, we can use the case analysis as an $\bot$_elimination operator. In the same way, if we study cases on the proof that $a \in A$ when A is isomorphic to absurdity (for example, when A is a set stating that zero is equal to successor n or that successor of n is less than zero, for a natural number n) we do not obtain any case to consider.

The following example shows how we can prove that a natural number n is less than a natural number m if we have a proof that m is less than zero.

```
ex2 ∈ (n, m ∈ N; <(m, 0)) <(n, m)
  ex2(n, m, h) ≡ case h ∈ <(m, 0) of
                 end
```

Here, h is a proof that m is less than zero. For each of the possible ways of constructing the proof h, we want to construct a proof that n is less than m. As there is no natural number less than zero, there is no case in the case analysis, and hence the proof of the thesis is trivial.

2.3 Working with ALF

All of the ALF definitions and proofs we present here and in later sections have been pretty printed by ALF itself. That is, all of them have been checked in ALF. In addition, we have made use of the layout facility of ALF that allows us to hide the declaration of some parameters, in the definitions of both sets and theorems. However, this has only been done when the hidden parameters do not contribute to the understanding of the definition. In general, the criterion used when hiding declarations is the following: when defining a function (or a set), we hide the declaration $a \in A$ if the parameter a occurs later on as part of the declaration of any other argument of the function (or the set), unless we perform pattern matching over the parameter a when defining the function.

In the following example, ex3 is a function that takes a proof that the natural number n is less than or equal to the natural number m into a proof that the successor of n is less than or equal to the successor of m.

```
ex3 ∈ (≤(n, m)) ≤(s(n), s(m))
```

Notice that the declarations $n \in \mathrm{N}$ and $m \in \mathrm{N}$ are hidden. However, they do not contribute to the understanding of the definition since we can easily infer the types of n and m from the facts that the relation $\leq$ and the function s are only defined for natural numbers.

In the next example, ex4 is a function that, given three natural numbers n, m and p, takes a proof that n is less than the addition of m and p into a proof that n is less than the addition of the successor of m and p.

```
ex4 ∈ (p ∈ N; <(n, +(m, p))) <(n, +(s(m), p))
```

Here, as the addition operator, the less-than relation and the function s are defined only for natural numbers, we can easily infer that the variables n, m and p should be natural numbers. Thus, we can hide the declarations of these three arguments. However, we need to perform pattern matching over the number p as part of the definition of the function ex4, and then we do not hide the declaration of p.

We now present some general set formers and constructors in Martin-Löf's type theory, together with some of their operators and properties.

2.3.1 Logical Constants

See section C.1 for the complete definitions of the following logical constants and their properties.

Absurdity: The set former is $\bot \in \mathbf{Set}$, and has no constructors.

And: Represents conjunction of two propositions. Here, we present the set former and its only constructor, and the type of the operators that select the first and second proposition of the conjunction. See section C.1.2 for the complete ALF code.

$\wedge \in (A, B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \wedge_{\mathrm{I}} \in (a \in A;\ b \in B)\ \wedge(A, B)$
$\mathrm{fst} \in (\wedge(A, B))\ A$
$\mathrm{snd} \in (\wedge(A, B))\ B$

Or: Represents disjunction of two propositions. Here, we present the set former and its two constructors.

$\vee \in (A, B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \vee_{\mathrm{L}} \in (a \in A)\ \vee(A, B)$
$\quad \vee_{\mathrm{R}} \in (b \in B)\ \vee(A, B)$

Imply: Represents implication between two propositions. Here, we present the set former and its only constructor, the type of the elimination operator associated with implication and the type of the transitivity property for implication. See section C.1.6 for the complete ALF code.

$\Rightarrow \in (A, B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \Rightarrow_{\mathrm{I}} \in (f \in (A)\, B)\ \Rightarrow(A, B)$
$\Rightarrow_{\mathrm{E}} \in (f \in \Rightarrow(A, B);\ a \in A)\ B$
$\Rightarrow_{\mathrm{trans}} \in (\Rightarrow(A, B);\ \Rightarrow(B, C))\ \Rightarrow(A, C)$

Equivalence: Now, we can present the definition of logical equivalence as the following abbreviation:

$\Leftrightarrow \in (A, B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \Leftrightarrow \equiv [A, B] \wedge(\Rightarrow(A, B), \Rightarrow(B, A))$

Not: Represents the negation operator. This operator is actually defined as an abbreviation.

$\neg \in (A \in \mathbf{Set})\ \mathbf{Set}$
$\quad \neg \equiv [A] \Rightarrow(A, \bot)$

Exists: Represents the existential quantifier. Here, we present the set former and its only constructor, and the type of the operators that select the element that satisfies the proposition and the proof that this particular element satisfies the proposition. See section C.1.5 for the complete ALF code.

$\exists \in (A \in \mathbf{Set};\ B \in (A)\,\mathbf{Set})\ \mathbf{Set}$
$\quad \exists_{\mathrm{I}} \in (a \in A;\ b \in B(a))\ \exists(A, B)$
$\mathrm{witness} \in (\exists(A, B))\ A$
$\mathrm{proof} \in (h \in \exists(A, B))\ B(\mathrm{witness}(h))$

Forall: Represents the universal quantifier. Here, we present the set former and its only constructor, and the type of the elimination operator associated with the universal quantifier. See section C.1.8 for the complete ALF code.

$\forall \in (A \in \mathbf{Set};\ B \in (A)\,\mathbf{Set})\ \mathbf{Set}$
$\quad \forall_{\mathrm{I}} \in (f \in (a \in A)\,B(a))\ \forall(A, B)$
$\forall_{\mathrm{E}} \in (\forall(A, B);\ a \in A)\ B(a)$

2.3.2 Some Useful General Predicates

See section C.2 for the complete definitions of the following predicates and their properties.

Identity: Represents propositional equality. Its only constructor states that an object is equal to itself. Together with the definition of the set, we prove the symmetry and transitivity properties, the congruence property with respect to functions of one and two arguments, and two substitutivity properties. Below, we present the definition of the predicate and the types of the mentioned properties. See section C.2.3 for the complete ALF code.

$= \in (a, b \in A)\ \mathbf{Set}$
$\quad \mathrm{refl} \in (a \in A)\ =(a, a)$
$=_{\mathrm{symm}} \in (=(a, b))\ =(b, a)$
$=_{\mathrm{trans}} \in (=(a, b);\ =(b, c))\ =(a, c)$
$=_{\mathrm{cong1}} \in (f \in (A)\,B;\ =(a_1, a_2))\ =(f(a_1), f(a_2))$
$=_{\mathrm{cong2}} \in (f \in (A; B)\,C;\ =(a_1, a_2);\ =(b_1, b_2))\ =(f(a_1, b_1), f(a_2, b_2))$
$=_{\mathrm{subst1}} \in (=(a, b);\ P(a))\ P(b)$
$=_{\mathrm{subst2}} \in (=(a, b);\ =(c, d);\ R(b, c))\ R(a, d)$

Although the following two predicates are not as general as the previous one, they play an important role in the following sections.

Accessibility: Represents the standard accessibility predicate, which is the standard way to handle general recursion in type theory (see [Acz77, Nor88]).

Given a set A, a binary relation $\prec$ on A and an element a in A, we can form the set $\mathrm{Acc}(A, \prec, a)$. This set is inhabited if, given a_i in A for $1 \leqslant i$, there exists no infinite descending sequence $\ldots \prec a_2 \prec a_1 \prec a$. If this is the case, we say that a is in the *well-founded part* of $\prec$ in A or that a is *accessible* by $\prec$ in A.

Constructively, we say that an element a in A is accessible if all elements smaller than a are accessible. In particular, if a is an initial element (that is, there is no x in A such that $x \prec a$), then a is accessible. This idea can be expressed by the following introduction rule for the accessibility predicate:

$$\frac{a \in A \qquad p \in (x \in A,\ h \in x \prec a)\mathrm{Acc}(A, \prec, x)}{\mathrm{acc}(a, p) \in \mathrm{Acc}(A, \prec, a)}$$

Notice that, in this way, we are able to capture the notion of infinite descending sequence in a single rule.

The elimination rule associated with the accessibility predicate, also known as the rule of well-founded recursion, is the following:

$$\frac{\begin{array}{c} a \in A \\ h \in \mathrm{Acc}(A, \prec, a) \\ e \in (x \in A,\ h' \in \mathrm{Acc}(A, \prec, x),\ p \in (y \in A,\ h_1 \in y \prec x)P(y))P(x) \end{array}}{\mathrm{wfrec}(a, h, e) \in P(a)}$$

and its computation rule is the following:

$$\mathrm{wfrec}(a, \mathrm{acc}(a, p), e) = e(a, \mathrm{acc}(a, p), [y, h]\mathrm{wfrec}(y, p(y, h), e)) \in P(a)$$

If all the elements in A are accessible by $\prec$, the set A is said to be *well-founded* by $\prec$, which is denoted by $\mathrm{WF}(A, \prec)$.

Below, we present the ALF definition of the accessibility predicate, the type of the well-foundedness predicate and the type of the elimination operator associated with the accessibility predicate.

```
Acc ∈ (A ∈ Set; less ∈ (A; A) Set; a ∈ A) Set
    acc ∈ (a ∈ A; (x ∈ A; less(x, a)) Acc(A, less, x)) Acc(A, less, a)
WF ∈ (A ∈ Set; less ∈ (A; A) Set) Set
wfrec ∈ (a ∈ A;
          Acc(A, less, a);
          e ∈ (x ∈ A; Acc(A, less, x); (y ∈ A; less(y, x))P(y)) P(x)
        ) P(a)
```

See section C.2.1 for the complete ALF definition of this predicate.

Decidability: We can think of this predicate as the set of decidable propositions. The set former has two constructors, depending on whether a proposition or its negation can be proven.

```
Dec ∈ (Set) Set
    yes ∈ (h ∈ P) Dec(P)
    no ∈ (h ∈ ¬(P)) Dec(P)
```

Observe that another possible way to define this predicate is the following:

```
Dec ∈ (Set) Set
    Dec ≡ [P]∨(P, ¬(P))
```

2.3.3 Some Useful General Data Types

See section C.3 for the complete definitions of the following data types and their properties.

List: The set of lists over a set A. Here, we present the set former and its two constructors, the type of the function that concatenates two lists and the type of the membership, non membership, disjoint and inclusion relations over lists.

List $\in$ ($A \in$ **Set**) **Set**
  [] $\in$ List(A)
  : $\in$ ($l \in$ List(A); $a \in A$) List(A)
++ $\in$ ($l_1, l_2 \in$ List(A)) List(A)
$\in_{\mathrm{L}}$ $\in$ ($a \in A$; $l \in$ List(A)) **Set**
$\notin_{\mathrm{L}}$ $\in$ ($a \in A$; $l \in$ List(A)) **Set**
Disjoint $\in$ ($l, l_1 \in$ List(A)) **Set**
$\subseteq$ $\in$ ($l_1, l_2 \in$ List(A)) **Set**

We also define several properties of lists. See section C.3.2 for the complete ALF definitions and properties of lists.

N: The set of natural numbers. In addition to the set of natural numbers, we define addition and the relations less-than and less-than or equal-to. Here, we present the set former and its two constructors, and the type of the addition and the two relations we mentioned above.

N $\in$ **Set**
  0 $\in$ N
  s $\in$ ($n \in$ N) N
+ $\in$ ($n, m \in$ N) N
< $\in$ ($n, m \in$ N) **Set**
$\leq$ $\in$ ($n, m \in$ N) **Set**

We also prove several properties of natural numbers. Among them we prove the decidability of the equality of natural numbers, the associativity and commutativity of addition, and we prove that N is well-founded by <. See section C.3.4 for the complete ALF definitions and properties of natural numbers.

Pair: The set of pairs over the sets A and B. Below, we give the set former and its only constructor.

Pair $\in$ ($A, B \in$ **Set**) **Set**
  . $\in$ ($a \in A$; $b \in B$) Pair(A, B)

Chapter 3

A Small Example

In this chapter, we present the formalisation of a small and general recursive algorithm in Martin-Löf's type theory: division-by-two over natural numbers. By using this very simple example, we illustrate the methodology we introduce in this work for writing general recursive algorithms in type theory. In addition, we show the advantages of this methodology over the standard way of defining general recursive algorithms in type theory, which is by using the accessibility predicate Acc.

Here, we follow the same process as the one we use in the next chapter to define the unification algorithm in type theory. That is, we first define the Haskell version of the algorithm. Then, we define the type theory version of the algorithm that uses the standard accessibility predicate Acc to handle the recursive calls and we point out the problems of this formalisation. Afterwards, we define a special-purpose accessibility predicate, which we call DivAcc, specifically defined for this case study. Intuitively, this predicate can be seen as defining the set of natural numbers on which the division-by-two algorithm terminates. Then, we show that actually all natural numbers satisfy this predicate, which means that the division-by-two algorithm terminates on all possible inputs. Finally, we write a new (and final) version of the division-by-two algorithm in type theory that is defined by structural recursion on the proof that the natural number to be divided satisfies the predicate DivAcc. We end this chapter by showing that the methodology we introduce here for writing the division-by-two algorithm in type theory admits a program-extraction process.

3.1 The Haskell Version of the Algorithm

In order to define the Haskell algorithm that divides a natural number by two, we first define the set of natural numbers in Haskell, the addition and subtraction operations (`<+>` and `<->` respectively) over natural numbers, and the less-than relation `<<` over natural numbers.

```
data Nat = Z | S Nat

one = S Z
two = S (S Z)

(<+>) :: Nat -> Nat -> Nat
n <+> Z     = n
n <+> (S m) = S(n <+> m)

(<->) :: Nat -> Nat -> Nat
n     <-> Z     = n
Z     <-> m     = Z
(S n) <-> (S m) = n <-> m

(<<) :: Nat -> Nat -> Bool
n     << Z     = False
Z     << (S m) = True
(S n) << (S m) = n << m
```

Now, the Haskell algorithm that divides a natural number by two can be defined as follows:

```
div2 :: Nat -> Nat
div2 n | n << two      = Z
       | not(n << two) = one <+> (div2 (n <-> two))
```

Here, we ignore efficiency aspects such as the fact that the expression `n << two` is computed twice.

It is easy to see that this is a total algorithm that terminates on all possible inputs. However, the recursive call is made on an argument that is not structurally smaller than the argument n, though the value of the argument $n - 2$ on which we perform the recursive call is less than n. The fact that the recursive call is made on a non-structurally smaller argument is of great importance when working in Martin-Löf's type theory since there is no direct way of formalising general recursive algorithms in the theory.

3.2 Using the Standard Accessibility Predicate for the Formalisation

Before introducing the type theory version of the algorithm `div2` that uses the standard accessibility predicate to handle the recursive call, we present the complete type theory definitions of the addition and subtraction operations, and the less-than relation over natural numbers. Recall that the definition of the set of natural numbers was already introduced in section 2.3.3.

$$
\begin{array}{lll}
 & - \in (n, m \in \mathrm{N})\ \mathrm{N} & \\
+ \in (n, m \in \mathrm{N})\ \mathrm{N} & -(n, 0) \equiv n & < \in (n, m \in \mathrm{N})\ \mathbf{Set} \\
\quad +(n, 0) \equiv n & \quad -(0, \mathrm{s}(m_1)) \equiv 0 & \quad <_0 \in (m \in \mathrm{N})\ <(0, \mathrm{s}(m)) \\
\quad +(n, \mathrm{s}(m_1)) \equiv \mathrm{s}(+(n, m_1)) & \quad -(\mathrm{s}(n_1), \mathrm{s}(m_1)) \equiv -(n_1, m_1) & \quad <_\mathrm{s} \in (<(n, m))\ <(\mathrm{s}(n), \mathrm{s}(m))
\end{array}
$$

In what follows, assume $1 \in \mathrm{N}$ and $2 \in \mathrm{N}$ to be defined as expected. In order to present the type theory version of the algorithm that uses the standard accessibility predicate to handle the recursive call, we need to define two lemmas. The first lemma shows that it is decidable whether or not a natural number is less than two. This lemma has the following type in ALF:

$$<2_{\mathrm{dec}} \in (n \in \mathrm{N})\ \mathrm{Dec}(<(n, 2))$$

The second lemma establishes that if the natural number n is not less than two, then the result of subtracting two from n is less than n. This lemma has the following type in ALF:

$$<\!-2 \in (n \in \mathrm{N};\ \neg(<(n, 2)))\ <(-(n, 2), n)$$

Now, we present the type theory version of the division-by-two algorithm that uses the standard accessibility predicate Acc to handle the recursive call.

$$
\begin{array}{l}
\mathrm{div2}_{\mathrm{acc}} \in (n \in \mathrm{N};\ \mathrm{Acc}(\mathrm{N}, <, n))\ \mathrm{N} \\
\quad \mathrm{div2}_{\mathrm{acc}}(n, \mathrm{acc}(\_, h_1)) \equiv \mathbf{case}\ <2_{\mathrm{dec}}(n) \in \mathrm{Dec}(<(n, 2))\ \mathbf{of} \\
\qquad\qquad\qquad\qquad \mathrm{yes}(h) \Rightarrow 0 \\
\qquad\qquad\qquad\qquad \mathrm{no}(h) \Rightarrow +(1, \mathrm{div2}_{\mathrm{acc}}(-(n, 2), h_1(-(n, 2), <\!-2(n, h)))) \\
\qquad\qquad\qquad \mathbf{end}
\end{array}
$$

This algorithm is defined by recursion on the proof that the natural number to be divided is accessible by $<$. To define the algorithm, we consider cases depending on whether or not n is less than two. If so, we return the value zero. Otherwise, we add one to the result of dividing $n - 2$ by two, which means that we have to call the algorithm recursively on the value $n - 2$, and we have to supply a proof that the value $n - 2$ is accessible, which is given by the expression $h_1(-(n, 2), <\!-2(n, h))$.

As the function $\mathrm{allacc}_{\mathrm{N}}$ gives us a proof that any natural number is accessible by $<$ [1], we can define the algorithm that divides any natural number by two as follows:

$$
\begin{array}{l}
\mathrm{Div2}_{\mathrm{acc}} \in (n \in \mathrm{N})\ \mathrm{N} \\
\quad \mathrm{Div2}_{\mathrm{acc}}(n) \equiv \mathrm{div2}_{\mathrm{acc}}(n, \mathrm{allacc}_{\mathrm{N}}(n))
\end{array}
$$

The main disadvantage of this formalisation of the algorithm is that we have to supply a proof that $n - 2$ is accessible by $<$ to the recursive call. This proof has no computational meaning and its only purpose is to serve as a structurally smaller expression on which to perform the recursion and, in this way, guarantee the termination of the division-by-two algorithm. As we already mentioned, the proof that $n - 2$ is accessible by $<$ is given by the expression $h_1(-(n, 2), <\!-2(n, h))$, which uses a proof that $n - 2$ is less than n. Notice that, even for such a small example, this accessibility proof distracts our attention from the actual division process and enlarges the code of the algorithm.

[1] See section C.3.4 for the proof that all natural numbers are accessible by $<$.

Our intention is to overcome this problem by defining a special-purpose accessibility predicate for the division-by-two algorithm, that we call DivAcc, which contains useful information that can help us to write a new (and final) type theory version of the algorithm.

3.3 Using a Special-Purpose Accessibility Predicate for the Formalisation

To construct this special-purpose accessibility predicate we ask ourselves the following question: on which inputs does the division-by-two algorithm terminate? To find the answer to this question, we inspect the Haskell version of the algorithm we presented at the beginning of this chapter, putting special attention on the input value, the conditions that should be satisfied in order to give a basic result or to perform a recursive call, and the value on which we perform the recursive call. We distinguish two cases:

- If the input number n is less than two, then the algorithm terminates since we return the value zero.
- If the input number n is not less than two, then the algorithm can only terminate on the input n if it terminates on the input $n - 2$.

Following this description, we define the inductive predicate DivAcc over natural numbers by means of the following introduction rules:

$$\frac{n < 2}{\mathrm{DivAcc}(n)}$$

$$\frac{\neg(n < 2) \qquad \mathrm{DivAcc}(n - 2)}{\mathrm{DivAcc}(n)}$$

This predicate can easily be formalised in ALF as follows:

```
DivAcc ∈ (n ∈ N) Set
    divacc<2 ∈ (<(n, 2)) DivAcc(n)
    divacc≥2 ∈ (¬(<(n, 2)); DivAcc(−(n, 2))) DivAcc(n)
```

Now, we prove that the division-by-two algorithm terminates on all possible inputs, that is, we prove that all natural numbers satisfy our special-purpose accessibility predicate DivAcc. In this proof, we use the fact that the argument on which the algorithm performs a recursive call is strictly smaller than the original argument. Hence, the division-by-two process should terminate on any input since the set of natural numbers is well-founded with respect to the less-than relation. In the proof that all natural numbers satisfy the predicate DivAcc, we use a few properties of the relation less-than over natural numbers whose proofs can be found in section C.3.4. In addition, we need to define the following auxiliary lemma:

```
<to⊥ ∈ (<(s(s(n)), 2)) ⊥
```

Below, we present the proof that all natural numbers satisfy the predicate DivAcc.

$$\begin{array}{l}
\mathrm{divacc}_{aux} \in (n \in \mathrm{N};\ \mathrm{Acc}(\mathrm{N}, <, n);\ f \in (m \in \mathrm{N}; <(m, n))\,\mathrm{DivAcc}(m))\ \mathrm{DivAcc}(n) \\
\quad \mathrm{divacc}_{aux}(0, h, f) \equiv \mathrm{divacc}_{<2}(<_{ssR}(0)) \\
\quad \mathrm{divacc}_{aux}(\mathrm{s}(0), h, f) \equiv \mathrm{divacc}_{<2}(<_{sR}(\mathrm{s}(0))) \\
\quad \mathrm{divacc}_{aux}(\mathrm{s}(\mathrm{s}(n)), h, f) \equiv \mathrm{divacc}_{\geq 2}(\Rightarrow_{\mathrm{I}}(<_{to}\bot), f(n, <_{ssR}(n))) \\
\mathrm{allDivAcc} \in (n \in \mathrm{N})\ \mathrm{DivAcc}(n) \\
\quad \mathrm{allDivAcc}(n) \equiv \mathrm{wfrec}(n, \mathrm{allacc}_{\mathrm{N}}(n), \mathrm{divacc}_{aux})
\end{array}$$

Now, we present the type theory version of the division-by-two algorithm that uses the predicate DivAcc to handle the recursive call.

$$\begin{array}{l}
\mathrm{div2} \in (n \in \mathrm{N};\ \mathrm{DivAcc}(n))\ \mathrm{N} \\
\quad \mathrm{div2}(n, \mathrm{divacc}_{<2}(h_1)) \equiv 0 \\
\quad \mathrm{div2}(n, \mathrm{divacc}_{\geq 2}(h_1, h_2)) \equiv +(1, \mathrm{div2}(-(n, 2), h_2))
\end{array}$$

This function is defined by structural recursion on the proof that the number to be divided by two satisfies the predicate DivAcc. To write the algorithm, we first perform pattern matching over the proof that the input number satisfies the predicate DivAcc. As a result of the pattern matching we obtain two equations, one for each of the introduction rules of the predicate. The first equation considers the case where n is less than two and h_1 is a proof of it. Then, we return the value zero. The second equation considers the case where n is not less than two. Here, h_1 is a proof that n is not less than two and h_2 is a proof that $n - 2$ satisfies the predicate DivAcc. Then, we have to add one to the result of dividing $n - 2$ by two, which means that we have to call the algorithm recursively on the value $n - 2$. To the recursive call we have to supply a proof that the value $n - 2$ satisfies the predicate DivAcc which is given by the argument h_2.

Finally, we can use the previous function and the fact that all natural numbers satisfy the predicate DivAcc to write the division-by-two algorithm.

$$\begin{array}{l}
\mathrm{Div2} \in (n \in \mathrm{N})\ \mathrm{N} \\
\quad \mathrm{Div2}(n) \equiv \mathrm{div2}(n, \mathrm{allDivAcc}(n))
\end{array}$$

Notice that, even for such a small example, the version of the algorithm that uses our special predicate

$$\begin{array}{l}
\mathrm{div2} \in (n \in \mathrm{N};\ \mathrm{DivAcc}(n))\ \mathrm{N} \\
\quad \mathrm{div2}(n, \mathrm{divacc}_{<2}(h_1)) \equiv 0 \\
\quad \mathrm{div2}(n, \mathrm{divacc}_{\geq 2}(h_1, h_2)) \equiv +(1, \mathrm{div2}(-(n, 2), h_2))
\end{array}$$

is slightly shorter and a bit more readable than the type theory version of the algorithm that is defined by using the predicate Acc

$$\begin{array}{l}
\mathrm{div2}_{acc} \in (n \in \mathrm{N};\ \mathrm{Acc}(\mathrm{N}, <, n))\ \mathrm{N} \\
\quad \mathrm{div2}_{acc}(n, \mathrm{acc}(_, h_1)) \equiv \textbf{case}\ <2_{dec}(n) \in \mathrm{Dec}(<(n, 2))\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathrm{yes}(h) \Rightarrow 0 \\
\qquad\qquad\qquad\qquad \mathrm{no}(h) \Rightarrow +(1, \mathrm{div2}_{acc}(-(n, 2), h_1(-(n, 2), <-2(n, h)))) \\
\qquad\qquad\qquad \textbf{end}
\end{array}$$

3.4 Towards Program Extraction

To end this chapter, we show that it is possible (and easy) to extract a Haskell algorithm from the type theory version of algorithm div2 which uses our special-purpose predicate to handle the recursive call. Notice that there are two kinds of parameters among those in the proof that a given natural number n satisfies the predicate DivAcc: the parameters that are proofs of certain conditions that should be satisfied in order to give a result or to perform a recursive call (the parameter h_1 in both equations) and the parameter that is a proof that the number to be divided by two in the recursive call satisfies the predicate DivAcc (the parameter h_2 in the second equation). In order to obtain a Haskell algorithm from the type theory algorithm div2, we translate each of the ALF equations of the algorithm into a Haskell equation. In the translation of each of the ALF equations, we throw away the expressions that are proofs that a certain number satisfies the predicate DivAcc and we keep the expressions that represent conditions to be satisfied as guards of the Haskell equation. Then, we would obtain the following Haskell algorithm:

```
div2 :: Nat -> Nat
div2 n | n << two       = Z
div2 n | not (n << two) = one <+> (div2 (n <-> two))
```

Observe that this algorithm is the same as the Haskell algorithm we presented in section 3.1 and that we used for defining the special-purpose predicate DivAcc. The reason for this similarity is that this program-extraction process can be seen as the inverse of the process that takes the Haskell version of the algorithm into the type theory version that uses the predicate DivAcc to handle the recursive calls.

In section 4.6, we describe the extraction of a Haskell program from the formalisation of the unification algorithm in type theory that uses a special-purpose accessibility predicate to handle the recursive calls in more detailed.

Chapter 4

The Unification Algorithm

Here, we first introduce the Haskell [JHe+99] version of the unification algorithm that we consider, and we give an informal explanation of its termination. After introducing the necessary definitions in ALF, we describe how we can write the unification algorithm in Martin-Löf's type theory by using the standard accessibility predicate, and we show why the use of this predicate to write the unification algorithm in type theory is not a good solution in our case. Then, we present a special-purpose accessibility predicate for the unification algorithm and we show how we can use this predicate to write the unification algorithm in Martin-Löf's type theory. With this particular predicate, the resulting algorithm is short and elegant. Finally, we discuss a methodology that allows us to extract a Haskell program from the formalisation of the unification algorithm that uses our special-purpose accessibility predicate to handle the recursive calls.

4.1 The Haskell Version of the Unification Algorithm

In this section, we present the Haskell version of the unification algorithm that we consider. In order to do that, we first have to introduce a few definitions.

To define the set Term of terms, we assume two (possibly infinite) sets: a set Var of variables and a set Fun of function symbols. These sets are such that for each pair of variables and each pair of functions, it is decidable whether or not they are equal. We use x and y to range over variables, and f and g to range over functions.

A term is either a variable or a function applied to a (possibly empty) list of terms. We use t (possibly primed or subscripted) to range over terms. We define the terms in Term by means of the following abstract syntax:

$$t ::= x \mid f(t_1, \ldots, t_n)$$

We use lt (possibly primed or subscripted) to range over lists of terms.

Once we have defined the set of terms, we define the set ListPT of lists of pairs of terms and the set Subst of substitutions. A list of pairs of terms is a list of pairs of the form (t_1, t_2). A substitution is a list of pairs of the form (x, t), that is, the left element of the pair is a variable and the right one is a term[1]. We use *lp* and *sb* (possibly primed or subscripted) to range over lists of pairs of terms and substitutions, respectively.

Given a substitution *sb* of the form $[(x_1, t_1), \ldots, (x_n, t_n)]$, the *domain* of the substitution is the set of variables $\{x_1, \ldots, x_n\}$ and the *range* of the substitution is the set of variables that occur in the terms $t_1, \ldots, t_n$. Given a term t, the result of *applying* *sb* to t is denoted by $sb(t)$ and it is defined as the parallel substitution of t_i for x_i in t, for $1 \leqslant i \leqslant n$. Given a list of pairs of terms *lp* and a substitution *sb*, we say that *sb* *unifies* *lp* or that *sb* is a *unifier* of *lp*, if for each pair of terms (t_1, t_2) in *lp* it holds that $sb(t_1) = sb(t_2)$. Finally, we say that a substitution *sb* is a most general unifier of a list of pairs of terms *lp* if *sb* is the most general substitution that unifies *lp*. In other words, *sb* is a most general unifier of *lp* if *sb* unifies *lp* and, for any other substitution sb' that also unifies *lp*, there exists a substitution sb_1 such that $sb'(t) = sb_1(sb(t))$, for all terms t.

The unification algorithm we consider here is a deterministic version of the first (non-deterministic) algorithm presented by Martelli and Montanari in [MM82]. Given a list of pairs of terms, the algorithm returns a substitution that unifies the list if such a substitution exists, or the special value `Nothing` if there is no such substitution. A similar algorithm is presented by Peter Hancock in chapter 9 of [PJ87], although the input of Hancock's algorithm is just a pair of terms and not a list of pairs of terms.

In figure 4.1, we present the Haskell version of the unification algorithm that we consider. Once again, we ignore efficiency aspects such as the fact that some expressions are computed twice. The functions `length`, `zip`, `elem`, `++`, `==` and `&&` are predefined functions in Haskell: `length` takes a list and returns the length of the list, `zip` takes two lists and returns a list of corresponding pairs, `elem` is the membership function over lists, `++` concatenates two lists, `==` is the equality function and `&&` is the boolean function and. These functions have the following types:

```
length :: [a] -> Int
zip    :: [a] -> [b] -> [(a,b)]
elem   :: Eq a => a -> [a] -> Bool
(++)   :: [a] -> [a] -> [a]
(==)   :: Eq a => a -> a -> Bool
(&&)   :: Bool -> Bool -> Bool
```

Notice that both the function `elem` and the function `==` require an equality relation over the type of the elements in the list and the type of the elements to be compared, respectively.

We now explain the functions `varsT`, `substLPT` and `substS`. The function `varsT` returns the list of variables in a term, and the functions `substLPT` and

[1] For the moment, we impose no restrictions on the variables that occur in the left hand side of the pairs of a substitution.

```
type Var   = Int
type Fun   = Int

data Term = Var Var | Fun Fun [Term]

type PairS  = (Var,Term)
type Subst  = [PairS]
type PairT  = (Term,Term)
type ListPT = [PairT]

unify_H :: ListPT -> Maybe Subst
unify_H lp = unify_h lp []

unify_h :: ListPT -> Subst -> Maybe Subst
unify_h []                     sb  = Just sb
unify_h ((Var x,Var y):lp)     sb
        | x == y                   = unify_h lp sb
unify_h ((Var x,t):lp)         sb
        | x `elem` (varsT t)       = Nothing
        | not(x `elem` (varsT t)) =
                unify_h (substLPT x t lp) ((x,t):(substS x t sb))
unify_h ((Fun f lt,Var x):lp) sb  =
                                   unify_h ((Var x,Fun f lt):lp) sb
unify_h ((Fun f lt1,Fun g lt2):lp) sb
        | f /= g || length lt1 /= length lt2 = Nothing
        | f == g && length lt1 == length lt2 =
                                     unify_h ((zip lt1 lt2)++lp) sb
```

Figure 4.1: Haskell Version of the Unification Algorithm

`substS` substitute a term for a variable in all the terms of a list of pairs of terms and a substitution, respectively. These functions have the following types:

```
varsT    :: Term -> [Var]
substLPT :: Var -> Term -> ListPT -> ListPT
substS   :: Var -> Term -> Subst -> Subst
```

The algorithm presented in figure 4.1 works as follows: given a list of pairs of terms, the function `unify_H` computes a substitution that unifies the list, if such a substitution exists, by using the auxiliary function `unify_h`.

The function `unify_h` takes two arguments: a list of pairs of terms lp and a substitution sb. Then, if the set of variables in lp and the set of variables in sb are disjoint, the substitution that results from the execution of `unify_h` lp sb will be the smallest extension of sb that unifies lp. As the first time the function `unify_h` is called, it is called with the empty substitution [], then the substitution that

results from the execution of `unify_H` lp will be a most general substitution that unifies the input list of pairs of terms. From the definition of the algorithms, it is relatively easy to see that every time the algorithm `unify_h` is executed, the condition on the sets of variables that occur in the list of pairs of terms to be unified and in the accumulated substitution is satisfied.

If the list of pairs of terms is empty, the function `unify_h` returns the (accumulated) substitution sb. Otherwise, we consider four cases depending on the form of the first pair of terms in the list of pairs of terms. These cases are exhaustive and mutually disjoint.

The first case we consider is when both of the terms in the first pair of the list are the (variable) term x. As both terms are already equal, we remove the first pair from the list and we continue finding a unifier for the rest of the list.

If, on the other hand, the left term of the first pair is the variable x and the right one is a term t (notice that here, we know that the terms x and t are different), we check whether or not the variable x belongs to the set of variables in the term t. If so, as the term t is different from the (variable) term x, it means that x is a proper subterm of t. As the relation "is a proper subterm of" is preserved under substitution application, given any substitution sb, then $sb(x)$ is a proper subterm of $sb(t)$. Thus, there exists no substitution that unifies x and t, and consequently there exists no substitution that unifies the original list of pairs of terms. Therefore, we finish the execution of the algorithm with the result `Nothing`. If the variable x does not belong to the variables in t, then any substitution that unifies the original list of pairs of terms should make x and t equal, so we add the pair (x, t) to the resulting substitution. Besides, we substitute t for x both in the list lp and in the substitution sb and call the unification algorithm recursively. In this way, we eliminate the variable x from lp and sb since x does not belong to the variables in t. Notice that now, x only occurs as the left hand side of the pair (x, t) just introduced to the accumulated substitution. As this process is performed every time we add a new pair to the resulting substitution, all the variables in the domain of the resulting substitution will be different from each other.

The next equation considers the case where the left term of the first pair is not a variable (that is, it is a function application) but the right one is. Here, we just swap the terms in this pair and call the unification algorithm recursively. It is easy to see that now, this first pair of the list is going to be handled by the third equation of the algorithm. Notice that we could have written an equivalent algorithm if we would have performed here the same kind of computation as in the previous equation. However, this would have made the algorithm and the different proofs a bit longer.

The last equation considers the case where both terms in the first pair of the list are function applications. Here, we check whether or not the functions in both terms are the same, which is done by checking that the function symbols and the length of both lists of terms are equal. If the functions are not equal, there does not exist any substitution that unifies the first pair of the list. Thus, there does not exist any substitution that unifies the original list of pairs of terms and we return the value `Nothing`. If both functions are equal, then the

first pair of terms of the list has the form $(f(t_1, \ldots, t_n),\ f(t'_1, \ldots, t'_n))$. Now, any substitution that unifies the original list should also unify this pair, and consequently it should unify t_i and t'_i, for $1 \leqslant i \leqslant n$. With the help of the function `zip`, we create the list of pairs of terms $[(t_1,\ t'_1), \ldots, (t_n,\ t'_n)]$ from the lists of terms lt_1 and lt_2. Then, we concatenate this list with the rest of the original list of pairs of terms and we call the unification algorithm recursively.

4.2 Termination of the Unification Algorithm

As one can see from the unification algorithm given in figure 4.1, the recursion performed in the algorithm is not always on structurally smaller arguments. Thus, there is no easy syntactic condition that guarantees the termination of the algorithm. Here, we show that our unification algorithm always terminates by defining a function that maps lists of pairs of terms into triples of natural numbers, and showing that in every recursive call the triple that corresponds to the list on which we perform the recursion is strictly smaller than the triple that corresponds to the original list of pairs of terms. This mapping, which we call $\text{LPT}_{\text{to}}\text{N3}$, is a simplification of the mapping F presented in [MM82] to show the termination of their (non-deterministic) algorithm.

N^3- The Set of Triples of Natural Numbers: Consider the set N of natural numbers and the inequality relation $<$ over N with its usual meaning[2]. We use n and m (possibly subscripted) to range over natural numbers.

Consider now the set N^3 of triples of natural numbers and the lexicographic order $<_{\text{N3}}$ over triples. In other words, if n_1, n_2 and n_3 are natural numbers then $(n_1,\ n_2,\ n_3)$ is an element of N^3. The lexicographic order over elements of N^3 is defined as the smallest relation such that:

$$\begin{array}{ll} (n_1,\ n_2,\ n_3) <_{\text{N3}} (m_1,\ m_2,\ m_3) & \text{if } n_1 < m_1 \\ (n_1,\ n_2,\ n_3) <_{\text{N3}} (n_1,\ m_2,\ m_3) & \text{if } n_2 < m_2 \\ (n_1,\ n_2,\ n_3) <_{\text{N3}} (n_1,\ n_2,\ m_3) & \text{if } n_3 < m_3 \end{array}$$

As the set of natural numbers is well-founded by $<$, it can be proven that the set of triples of natural numbers, that is N^3, is well-founded by $<_{\text{N3}}$.

See section C.3.5 for the complete ALF definition of N^3 together with the ALF proof that N^3 is well-founded by $<_{\text{N3}}$.

The Function $\text{LPT}_{\text{to}}\text{N3}$: To define the function $\text{LPT}_{\text{to}}\text{N3}$ that maps lists of pairs of terms into triples of natural numbers, we use three auxiliary functions that take a list of pairs of terms and return a natural number. The first function, called $\#\text{vars}_{\text{LPT}}$, takes a list of pairs of terms and returns the number of different variables that occur in the list. The second function, called $\#\text{funs}_{\text{LPT}}$, takes a list of pairs of terms and returns the number of function applications that occur

[2]See section C.3.4 for the ALF definitions of the set N and its inequality $<$, and the ALF proof that N is well-founded by $<$.

in the list. The last function, called $\#\mathsf{eqs}_{\mathrm{LPT}}$, takes a list of pairs of terms and counts the number of pairs of the form (x, x) or $(f(lt), x)$ that appear in the list.

We now define the function $\mathrm{LPT_{to}N3}$ as follows:

$$\begin{aligned}&\mathrm{LPT_{to}N3} :: \mathsf{ListPT} \to \mathrm{N}^3\\ &\mathrm{LPT_{to}N3}(lp) = (\#\mathsf{vars}_{\mathrm{LPT}}(lp), \#\mathsf{funs}_{\mathrm{LPT}}(lp), \#\mathsf{eqs}_{\mathrm{LPT}}(lp))\end{aligned}$$

To show that the unification algorithm terminates, it is sufficient to show that, in every recursive call of the algorithm, we decrease the complexity measure of the list of pairs of terms to be unified. The measures we consider here are triples of natural numbers and the mapping from lists of pairs of terms into triples of natural numbers is the function $\mathrm{LPT_{to}N3}$. Hence, we have to show that the following inequalities hold:

$$\begin{aligned}\mathrm{LPT_{to}N3}(lp) &<_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ x) : lp)\\ \mathrm{LPT_{to}N3}(lp\,[x{:=}t]) &<_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ t) : lp) \qquad \text{if } x \notin_{\mathrm{L}} \mathsf{vars}_{\mathrm{T}}(t)\\ \mathrm{LPT_{to}N3}((x,\ f(lt)) : lp) &<_{\mathrm{N3}} \mathrm{LPT_{to}N3}((f(lt),\ x) : lp)\\ \mathrm{LPT_{to}N3}((\mathsf{zip}\ lt_1\ lt_2) +\!\!+ lp) &<_{\mathrm{N3}} \mathrm{LPT_{to}N3}((f(lt_1),\ f(lt_2)) : lp)\end{aligned}$$

where $lp\,[x{:=}t]$ denotes the function that substitutes the term t for the variable x in the list of pairs of terms lp, $\mathsf{vars}_{\mathrm{T}}$ is the function that returns the set of variables in a term and the function $\notin_{\mathrm{L}}$ is the non-membership relation over lists (these functions correspond to the functions `substL`, `varsT` and the negation of the function `elem` respectively, in the algorithm of figure 4.1). See appendix A for a discussion of the ALF proofs of these inequalities and section C.4.4 for the complete ALF code of the formalisation of these proofs.

Informal Proof of $\mathrm{LPT_{to}N3}(lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ x) : lp)$**:** If the variable x does not occur in the list lp we have that $\#\mathsf{vars}_{\mathrm{LPT}}(lp) < \#\mathsf{vars}_{\mathrm{LPT}}((x,\ x) : lp)$, and consequently we know that $\mathrm{LPT_{to}N3}(lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ x) : lp)$ by the first inequality in the definition of $<_{\mathrm{N3}}$. Otherwise, $\#\mathsf{vars}_{\mathrm{LPT}}(lp) = \#\mathsf{vars}_{\mathrm{LPT}}((x,\ x) : lp)$. Here, $\#\mathsf{funs}_{\mathrm{LPT}}(lp) = \#\mathsf{funs}_{\mathrm{LPT}}((x,\ x) : lp)$ and, since the pair $(x,\ x)$ is one of the pairs counted by the function $\#\mathsf{eqs}_{\mathrm{LPT}}$, we know that $\#\mathsf{eqs}_{\mathrm{LPT}}(lp) < \#\mathsf{eqs}_{\mathrm{LPT}}((x,\ x) : lp)$. Thus, $\mathrm{LPT_{to}N3}(lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ x) : lp)$ by the third inequality in the definition of $<_{\mathrm{N3}}$.

Informal Proof of $\mathrm{LPT_{to}N3}(lp\,[x{:=}t]) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ t) : lp)$**:** As $x \notin_{\mathrm{L}} \mathsf{vars}_{\mathrm{T}}(t)$, x does not belong to the set of variables of the list $lp\,[x{:=}t]$. Hence, we have that $\#\mathsf{vars}_{\mathrm{LPT}}(lp\,[x{:=}t]) < \#\mathsf{vars}_{\mathrm{LPT}}((x,\ t) : lp)$ and thus, by the first inequality in the definition of $<_{\mathrm{N3}}$, we have that $\mathrm{LPT_{to}N3}(lp\,[x{:=}t]) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((x,\ t) : lp)$.

Informal Proof of $\mathrm{LPT_{to}N3}((x,\ f(lt)) : lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((f(lt),\ x) : lp)$**:** Here, we know that $\#\mathsf{vars}_{\mathrm{LPT}}((x,\ f(lt)) : lp) = \#\mathsf{vars}_{\mathrm{LPT}}((f(lt),\ x) : lp)$ and we also know that $\#\mathsf{funs}_{\mathrm{LPT}}((x,\ f(lt)) : lp) = \#\mathsf{funs}_{\mathrm{LPT}}((f(lt),\ x) : lp)$. Since the pair $(f(lt),\ x)$ is one of the pairs counted by the function $\#\mathsf{eqs}_{\mathrm{LPT}}$, we can easily show that

$\#\mathrm{eqs}_{\mathrm{LPT}}((x,\ f(lt)) : lp) < \#\mathrm{eqs}_{\mathrm{LPT}}((f(lt),\ x) : lp)$. Thus, by the third inequality in the definition of $<_{\mathrm{N3}}$, we have $\mathrm{LPT_{to}N3}((x,\ f(lt)) : lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((f(lt),\ x) : lp)$.

Informal Proof of $\mathrm{LPT_{to}N3}((\mathsf{zip}\ lt_1\ lt_2) +\!\!+ lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((f(lt_1),\ f(lt_2)) : lp)$**:** Here, we know that $\#\mathrm{vars}_{\mathrm{LPT}}((\mathsf{zip}\ lt_1\ lt_2) +\!\!+ lp) = \#\mathrm{vars}_{\mathrm{LPT}}((f(lt_1),\ f(lt_2)) : lp)$. We also know that $\#\mathrm{funs}_{\mathrm{LPT}}((\mathsf{zip}\ lt_1\ lt_2) +\!\!+ lp) < \#\mathrm{funs}_{\mathrm{LPT}}((f(lt_1),\ f(lt_2)) : lp)$ because the two applications of f in the pair $(f(lt_1),\ f(lt_2))$ do not occur in the list of pairs of terms $(\mathsf{zip}\ lt_1\ lt_2)$. Hence, by the second inequality in the definition of $<_{\mathrm{N3}}$, $\mathrm{LPT_{to}N3}((\mathsf{zip}\ lt_1\ lt_2) +\!\!+ lp) <_{\mathrm{N3}} \mathrm{LPT_{to}N3}((f(lt_1),\ f(lt_2)) : lp)$.

4.3 Terms, Lists of Pairs of Terms and Substitutions in ALF

In this section, we present the representation of terms, lists of pairs of terms and substitutions in Martin-Löf's type theory.

In order to formalise the set Term of terms, we use the notion of vectors of a certain length instead of the lists used in the Haskell version of the algorithm. In this way, we have information about the length of a list as part of its type.

A vector is either empty and has length 0, or it has length $n + 1$ and is formed from adding an element to a vector of length n. Hence, the length of a vector is part of its type. Vectors are formalised in type theory as follows:

Vector ∈ (n ∈ N; A ∈ **Set**) **Set**
  $[]_{\mathrm{v}}$ ∈ Vector(0, A)
  $:_{\mathrm{v}}$ ∈ (v ∈ Vector(n, A); a ∈ A) Vector(s(n), A)

As the sets Var of variables and Fun of function symbols we use the set of natural numbers. Then, the decidability of the equality of variables and function symbols becomes the decidability of equality of natural numbers.

Var ∈ **Set**
  Var ≡ N
$\mathrm{Var_{dec}}$ ∈ (x, y ∈ Var) Dec(=(x, y))

Fun ∈ **Set**
  Fun ≡ N
$\mathrm{Fun_{dec}}$ ∈ (f, g ∈ Fun) Dec(=(f, g))

The following lemma allows us to choose between the elements a and b, depending on whether we have a proof that the variables x and y are equal or different.

$\mathrm{Var_{to}A}$ ∈ (a, b ∈ A; Dec(=(x, y))) A
  $\mathrm{Var_{to}A}(a, b, \mathrm{yes}(h_1)) \equiv a$
  $\mathrm{Var_{to}A}(a, b, \mathrm{no}(h_1)) \equiv b$

Terms and vectors of terms are defined in ALF as follows:

Term ∈ **Set**
  var ∈ (x ∈ Var) Term
  fun ∈ (f ∈ Fun; lt ∈ Vector(n, Term)) Term
VTerm ∈ (n ∈ N) **Set**
  VTerm(n) ≡ Vector(n, Term)

The ALF representation of parametric lists and pairs was already introduced in section 2.3.3. We now use them to define the set ListPT of lists of pairs of terms and the set Subst of substitutions:

```
PairT ∈ Set                    PairS ∈ Set
    PairT ≡ Pair(Term, Term)       PairS ≡ Pair(Var, Term)
ListPT ∈ Set                   Subst ∈ Set
    ListPT ≡ List(PairT)           Subst ≡ List(PairS)
```

Notice that, as part of the definition of substitutions, we do not require the variables in the domain of a substitution to be different from each other. Furthermore, a variable may occur in its associated term. In this way, the definition of substitutions remains simple, which makes it easier to prove lemmas about substitutions. We impose these two conditions in the definition of idempotence in section 5.2.

In ALF, we write $:=_{T}$, $:=_{VT}$, $:=_{LPT}$ and $:=_{S}$ for the functions that substitute a term for a variable in a term, in a vector of terms, in a list of pairs of terms and in a substitution respectively, and we write vars_{T}, vars_{VT}, vars_{LPT} and vars_{S} for the functions that return the list of variables that occur in a term, in a vector of terms, in a list of pairs of terms and in a substitution respectively.

See sections C.4.1, C.4.3 and C.4.5 for the complete ALF definitions of terms and vectors of terms, lists of pairs of terms and substitutions respectively.

To finish this section, we present the type of the ALF lemmas that correspond to the four inequalities over lists of pairs of terms presented in the previous section.

```
<LPTvar_var ∈ (x ∈ Var; lp ∈ ListPT) <N3(LPTtoN3(lp), LPTtoN3(:(lp, .(var(x), var(x)))))
<LPT:=var_term ∈ (lp ∈ ListPT;
                  ∉L(x, varsT(t))
                 )<N3(LPTtoN3(:=LPT(x, t, lp)), LPTtoN3(:(lp, .(var(x), t))))
<LPTvar_fun ∈ (f ∈ Fun;
               x ∈ Var;
               lt ∈ VTerm(n);
               lp ∈ ListPT
              )<N3(LPTtoN3(:(lp, .(var(x), fun(f, lt)))), LPTtoN3(:(lp, .(fun(f, lt), var(x)))))
<LPTzip_fun_fun ∈ (f, g ∈ Fun;
                   lt1, lt2 ∈ VTerm(n);
                   lp ∈ ListPT
                  )<N3(LPTtoN3(++(zip(lt1, lt2), lp)), LPTtoN3(:(lp, .(fun(f, lt1), fun(g, lt2)))))
```

See appendix A for a discussion of the ALF proofs of these inequalities and section C.4.4 for the complete ALF code of the formalisation of these proofs.

4.4 The Unification Algorithm in Type Theory: First Attempt

In this section, we present our first attempt to formalise the unification algorithm in Martin-Löf's type theory. This formalisation uses the standard accessibility predicate to handle the recursive calls. Recall that the definition of the

standard accessibility predicate Acc was already introduced in section 2.3.2.

4.4.1 The Unification Algorithm using the Accessibility Predicate

In order to write the type theory version of the unification algorithm that uses the standard accessibility predicate to handle the recursive calls, we define a binary relation $<_{LPT}$ over lists of pairs of terms as follows:

$<_{LPT} \in (lp_1, lp_2 \in \text{ListPT})$ **Set**
$\quad <_{LPT} \in (<_{N3}(\text{LPT}_{to}\text{N3}(lp_1), \text{LPT}_{to}\text{N3}(lp_2))) <_{LPT}(lp_1, lp_2)$

As the set N^3 is well-founded by $<_{N3}$, it is easy to prove that the set ListPT is well-founded by $<_{LPT}$. Then, we prove the following lemma in ALF:

$\text{allacc}_{LPT} \in (lp \in \text{ListPT})\ \text{Acc}(\text{ListPT}, <_{LPT}, lp)$

that given a list of pairs of terms returns a proof that the list is accessible by $<_{LPT}$.

Below, we explain the necessary steps we perform in order to write the algorithm Unify_{acc}, which is the type theory version of the unification algorithm that uses the accessibility predicate to handle the recursive calls.

Instead of the `Maybe` type of Haskell, we use here the logic connective $\vee$ defined in section 2.3.1 and a set Error defined in ALF as follows:

Error $\in$ **Set**
$\quad$ error $\in$ Error

Given a list of pairs of terms lp, the unification algorithm Unify_{acc} returns either a substitution that unifies lp or the value error, if there does not exist such substitution. As in the Haskell version, the algorithm Unify_{acc} calls the algorithm unify_{acc} with the list lp and the empty substitution, but now it also supplies a proof that the list lp is accessible by $<_{LPT}$. We have:

$\text{Unify}_{acc} \in (lp \in \text{ListPT}) \vee(\text{Subst}, \text{Error})$
$\quad \text{Unify}_{acc}(lp) \equiv \text{unify}_{acc}(lp, [], \text{allacc}_{LPT}(lp))$

The algorithm unify_{acc} is defined by recursion on the proof that the input list is accessible by $<_{LPT}$. By performing pattern matching on the proof that the list lp is accessible by $<_{LPT}$, we obtain the following (incomplete) ALF code:

$\text{unify}_{acc} \in (lp \in \text{ListPT};\ sb \in \text{Subst};\ \text{Acc}(\text{ListPT}, <_{LPT}, lp)) \vee(\text{Subst}, \text{Error})$
$\quad \text{unify}_{acc}(lp, sb, \text{acc}(\_, h_1)) \equiv ?_{\text{unify\_acc.0.0.E}}$

where h_1 is the function that takes a list lp' and a proof that lp' is smaller than lp, and returns a proof that lp' is accessible by $<_{LPT}$.

In order to obtain the cases we are interested in, we have to perform a few pattern matchings on the list lp and a few case analyses. For the case analyses, we use the following decidability lemmas:

$\text{Var}_{dec} \in (x, y \in \text{Var})\ \text{Dec}(=(x, y))$
$\in_{dec} \in (x \in \text{Var};\ l \in \text{ListVar})\ \text{Dec}(\in_{L}(x, l))$
$\text{Fun}_{dec} \in (f, g \in \text{Fun})\ \text{Dec}(=(f, g))$
$\text{N}_{dec} \in (n, m \in \text{N})\ \text{Dec}(=(n, m))$

After filling in the basic results, we obtain the following incomplete algorithm in ALF:

```
unify_acc ∈ (lp ∈ ListPT; sb ∈ Subst; Acc(ListPT, <_LPT, lp)) ∨(Subst, Error)
  unify_acc([], sb, acc(_, h1)) ≡ ∨_L(sb)
  unify_acc(:(lp1, .(var(x), var(x1))), sb, acc(_, h1)) ≡
    case Var_dec(x, x1) ∈ Dec(=(x, x1)) of
      yes(refl(_)) ⇒ ?unify_acc.1.0.E
      no(h) ⇒ ?unify_acc.1.1.E
    end
  unify_acc(:(lp1, .(var(x), fun(f, lt))), sb, acc(_, h1)) ≡
    case ∈_dec(x, vars_T(fun(f, lt))) ∈ Dec(∈_L(x, vars_T(fun(f, lt)))) of
      yes(h) ⇒ ∨_R(error)
      no(h) ⇒ ?unify_acc.2.1.E
    end
  unify_acc(:(lp1, .(fun(f, lt), var(x))), sb, acc(_, h1)) ≡ ?unify_acc.3.E
  unify_acc(:(lp1, .(fun(f1, lt1), fun(f2, lt2))), sb, acc(_, h1)) ≡
    case Fun_dec(f1, f2) ∈ Dec(=(f1, f2)) of
      yes(refl(_)) ⇒ case N_dec(n1, n2) ∈ Dec(=(n1, n2)) of
                       yes(refl(_)) ⇒ ?unify_acc.4.1.0.E
                       no(h2) ⇒ ∨_R(error)
                     end
      no(h) ⇒ ∨_R(error)
    end
```

Now, it only remains to fill in the cases where the recursive calls are performed. In the recursive calls, the fields that correspond to the lists of pairs of terms and the substitutions are the same as in the Haskell version of the algorithm. In addition, we have to supply proofs that the new lists to be unified are accessible. To obtain these proofs, we use the function h_1. In each of the recursive calls, to the function h_1 we have to supply a proof that the new list to be unified is smaller than the original list. We use the lemmas presented in section 4.2 (see appendix A for the ALF proofs of the lemmas) for the proofs of the inequalities that we supply to the function h_1.

In figure 4.2, we present the complete formalisation of the unification algorithm in Martin-Löf's type theory that uses the standard accessibility predicate to handle the recursive calls.

4.4.2 Problems of this Formalisation

If we compare the algorithms in figures 4.1 and 4.2, it is easy to see that the latter is almost three times longer than the former. The longer the algorithm, the more difficult is to read and understand it and this, of course, creates an important gap between programming in a Haskell-like programming language and programming in Martin-Löf's type theory.

While the Haskell version of the algorithm contains only the necessary information for performing the computations, the type theory version of the algorithm needs extra information in order to handle the recursive calls. In the algorithm of figure 4.2, each recursive call has the form $\mathsf{unify}_{\mathsf{acc}}(lp', sb', h_1(lp', p))$ for a list of pairs of terms lp', a substitution sb' and a proof p that the list lp'

$\mathrm{Unify}_{acc} \in (lp \in \mathrm{ListPT}) \vee(\mathrm{Subst}, \mathrm{Error})$
$\quad \mathrm{Unify}_{acc}(lp) \equiv \mathrm{unify}_{acc}(lp, [], \mathrm{allacc}_{LPT}(lp))$

$\mathrm{unify}_{acc} \in (lp \in \mathrm{ListPT};\ sb \in \mathrm{Subst};\ \mathrm{Acc}(\mathrm{ListPT}, <_{LPT}, lp)) \vee(\mathrm{Subst}, \mathrm{Error})$
$\quad \mathrm{unify}_{acc}([], sb, \mathrm{acc}(\_, h_1)) \equiv \vee_L(sb)$
$\quad \mathrm{unify}_{acc}(:(lp_1, .(\mathrm{var}(x), \mathrm{var}(x_1))), sb, \mathrm{acc}(\_, h_1)) \equiv$
$\quad\quad$ **case** $\mathrm{Var}_{dec}(x, x_1) \in \mathrm{Dec}(=(x, x_1))$ **of**
$\quad\quad\quad \mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow \mathrm{unify}_{acc}(lp_1, sb, h_1(lp_1, <_{LPT}(<_{LPTvar\_var}(x_1, lp_1))))$
$\quad\quad\quad \mathrm{no}(h) \Rightarrow$
$\quad\quad\quad\quad \mathrm{unify}_{acc}(:=_{LPT}(x, \mathrm{var}(x_1), lp_1),$
$\quad\quad\quad\quad\quad :(:=_S(x, \mathrm{var}(x_1), sb), .(x, \mathrm{var}(x_1))),$
$\quad\quad\quad\quad\quad h_1(:=_{LPT}(x, \mathrm{var}(x_1), lp_1), <_{LPT}(<_{LPT:=var\_term}(lp_1, \notin:(\notin[])(x), h)))))$
$\quad\quad$ **end**
$\quad \mathrm{unify}_{acc}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f, lt))), sb, \mathrm{acc}(\_, h_1)) \equiv$
$\quad\quad$ **case** $\in_{dec}(x, \mathrm{vars}_T(\mathrm{fun}(f, lt))) \in \mathrm{Dec}(\in_L(x, \mathrm{vars}_T(\mathrm{fun}(f, lt))))$ **of**
$\quad\quad\quad \mathrm{yes}(h) \Rightarrow \vee_R(\mathrm{error})$
$\quad\quad\quad \mathrm{no}(h) \Rightarrow$
$\quad\quad\quad\quad \mathrm{unify}_{acc}($
$\quad\quad\quad\quad\quad :=_{LPT}(x, \mathrm{fun}(f, lt), lp_1),$
$\quad\quad\quad\quad\quad :(:=_S(x, \mathrm{fun}(f, lt), sb), .(x, \mathrm{fun}(f, lt))),$
$\quad\quad\quad\quad\quad h_1(:=_{LPT}(x, \mathrm{fun}(f, lt), lp_1), <_{LPT}(<_{LPT:=var\_term}(lp_1, \neg\in_{to}\notin(\mathrm{vars}_T(\mathrm{fun}(f, lt)), h)))))$
$\quad\quad$ **end**
$\quad \mathrm{unify}_{acc}(:(lp_1, .(\mathrm{fun}(f, lt), \mathrm{var}(x))), sb, \mathrm{acc}(\_, h_1)) \equiv$
$\quad\quad \mathrm{unify}_{acc}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f, lt))),$
$\quad\quad\quad sb,$
$\quad\quad\quad h_1(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f, lt))), <_{LPT}(<_{LPTvar\_fun}(f, x, lt, lp_1))))$
$\quad \mathrm{unify}_{acc}(:(lp_1, .(\mathrm{fun}(f_1, lt_1), \mathrm{fun}(f_2, lt_2))), sb, \mathrm{acc}(\_, h_1)) \equiv$
$\quad\quad$ **case** $\mathrm{Fun}_{dec}(f_1, f_2) \in \mathrm{Dec}(=(f_1, f_2))$ **of**
$\quad\quad\quad \mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow$
$\quad\quad\quad\quad$ **case** $\mathrm{N}_{dec}(n_1, n_2) \in \mathrm{Dec}(=(n_1, n_2))$ **of**
$\quad\quad\quad\quad\quad \mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow$
$\quad\quad\quad\quad\quad\quad \mathrm{unify}_{acc}(++(\mathrm{zip}(lt_1, lt_2), lp_1),$
$\quad\quad\quad\quad\quad\quad\quad sb,$
$\quad\quad\quad\quad\quad\quad\quad h_1(++(\mathrm{zip}(lt_1, lt_2), lp_1), <_{LPT}(<_{LPTzip\_fun\_fun}(f_2, f_2, lt_1, lt_2, lp_1))))$
$\quad\quad\quad\quad\quad \mathrm{no}(h_2) \Rightarrow \vee_R(\mathrm{error})$
$\quad\quad\quad\quad$ **end**
$\quad\quad\quad \mathrm{no}(h) \Rightarrow \vee_R(\mathrm{error})$
$\quad\quad$ **end**

Figure 4.2: Formalisation of the Unification Algorithm by using the Accessibility Predicate

is smaller than the original list. Notice that the list lp' appears twice in each recursive call, which implies having redundant information. In addition, the argument $h_1(lp', p)$ is computationally irrelevant; its only purpose is to serve as a structurally smaller argument on which to perform the recursion. Moreover, the proof p is usually long, which contributes to make the reading of the type theory version of the algorithm more difficult.

Most of these problems arise from the fact that the standard accessibility predicate is a general predicate, and then it has no particular information that can be of use in our specific case study. In the next section, we overcome the problems described above by presenting a special-purpose accessibility predicate for the unification algorithm. Our special-purpose predicate contains useful information for our specific case study, which allows us to do the recursive calls in the definition of the unification algorithm in a simple way. In this way, we obtain a formalisation of the algorithm that is short and elegant.

4.5 The Unification Algorithm in Type Theory: Second Attempt

In this section, we present the second (and final) attempt to formalise the unification algorithm in Martin-Löf's type theory. This formalisation uses a special-purpose accessibility predicate, which we call UniAcc and it is specially defined for this case study, to handle the recursive calls.

4.5.1 The UniAcc Predicate

Intuitively, we can think of this predicate as defining the set of lists of pairs of terms on which our unification algorithm terminates. In other words, a list of pairs of terms lp satisfies the predicate UniAcc if our algorithm terminates on the input list lp. Observe that, if for the input list lp the unification algorithm performs a recursive call on the list lp', the unification algorithm can only terminate on the input lp if it terminates on the input lp'. Then, a proof that the list lp' satisfies the special-purpose accessibility predicate is a requirement for the list lp to satisfy the predicate.

To define this predicate, we study the equations in the definition of the Haskell version of the algorithm `unify_h`, putting the emphasis on the input list, the lists on which we perform the recursion (if any) and any extra conditions (if any) that should be satisfied in order to produce a result or to perform a recursive call. We identify seven cases:

- If the input list is empty, then the algorithm terminates with a substitution.

- If the input list is of the form $(x, x) : lp$, then the algorithm can only terminate on the input list if it terminates on the list lp.

- If the input list is of the form $(x,\ t) : lp$ with $x \in \mathsf{vars_T}(t)$ and $x \neq t$, then the algorithm terminates since there does not exist a unifier for the input list.
- If the input list is of the form $(x,\ t) : lp$ with $x \notin_{\mathsf{L}} \mathsf{vars_T}(t)$, then the algorithm can only terminate on the input list if it terminates on the list $lp\,[x{:=}t]$.
- If the input list is of the form $(f(lt),\ x) : lp$, then the algorithm can only terminate on the input list if it terminates on the list $(x,\ f(lt)) : lp$.
- If the input list is of the form $(f(lt_1),\ g(lt_2)) : lp$ and it holds that $f \neq g$ or that $\mathsf{length}(lt_1) \neq \mathsf{length}(lt_2)$, then the algorithm terminates since there does not exist a possible unifier for the input list.
- If the input list is of the form $(f(lt_1),\ g(lt_2)) : lp$ and it holds that $f = g$ and that $\mathsf{length}(lt_1) = \mathsf{length}(lt_2)$, then the algorithm can only terminate on the input list if it terminates on the list $(\mathsf{zip}\ lt_1\ lt_2) +\!\!+ lp$.

Notice that these seven cases are exhaustive and mutually disjoint. Observe also that the condition $x \neq t$ that we added in the third case is not necessary in the Haskell version of the algorithm due to the way Haskell processes the equations that define an algorithm.

For each of these seven cases, we define an introduction rule for the predicate UniAcc. Each introduction rule contains the information we detailed above for the corresponding case. Then, each introduction rule has one of the following two patterns:

$$\frac{c_1 \ \cdots \ c_n}{\mathsf{UniAcc}(lp)} \qquad\qquad \frac{c_1 \ \cdots \ c_n \qquad \mathsf{UniAcc}(lp')}{\mathsf{UniAcc}(lp)}$$

where c_i, for $1 \leqslant i \leqslant n$, are the extra conditions that should be satisfied in order to produce a result or to perform a recursive call, lp' is the list on which we perform the recursion and lp is the input list. Thus, the introduction rules for the cases where we perform a recursive call follow the pattern of the right rule above, while the introduction rules for the other cases follow the pattern of the left rule above. Notice that we do not need to mention the substitutions nor the basic results of the algorithm in the introduction rules.

In figure 4.3, we present the definition of the inductive predicate UniAcc using introduction rules and its ALF formalisation. The premises of the form $a \neq b$ in the introduction rules were formalised as $\neg(=(a, b))$ in the ALF definition. Observe that in the last three constructors of the ALF formalisation, the lengths of the lists of terms are given as part of their types, that is, the lists of terms are declared as vectors of terms. In addition, as the declarations of the vectors of terms do not play an important role we can often hide them and then, by making the (visible part of the) definition of the constructors shorter, we contribute to a better understanding of the definition of the predicate. In the formalisation of the last introduction rule, the Haskell function `zip` is not exactly the same

Definition of the UniAcc Predicate using Introduction Rules

$$\frac{}{\text{UniAcc}([\,])}$$

$$\frac{\text{UniAcc}(lp)}{\text{UniAcc}((x,\ x) : lp)}$$

$$\frac{x \in \text{vars}_{\text{T}}(t) \qquad x \neq t}{\text{UniAcc}((x,\ t) : lp)}$$

$$\frac{x \notin_{\text{L}} \text{vars}_{\text{T}}(t) \qquad \text{UniAcc}(lp\,[x{:=}t])}{\text{UniAcc}((x,\ t) : lp)}$$

$$\frac{\text{UniAcc}((x,\ f(lt)) : lp)}{\text{UniAcc}((f(lt),\ x) : lp)}$$

$$\frac{f \neq g \ \vee\ \text{length}(lt_1) \neq \text{length}(lt_2)}{\text{UniAcc}((f(lt_1),\ g(lt_2)) : lp)}$$

$$\frac{\text{length}(lt_1) = \text{length}(lt_2) \qquad \text{UniAcc}((\text{zip}\ \ lt_1\ \ lt_2) +\!\!+ lp)}{\text{UniAcc}((f(lt_1),\ f(lt_2)) : lp)}$$

ALF Definition of the UniAcc Predicate

```
UniAcc ∈ (lp ∈ ListPT) Set
    uniacc[] ∈ UniAcc([])
    uniacc_var_var ∈ (x ∈ Var;
                      UniAcc(lp)
                     ) UniAcc(:(lp, .(var(x), var(x))))
    uniacc_var_term ∈ (lp ∈ ListPT;
                       ∈L(x, vars_T(t));
                       ¬(=(var(x), t))
                      ) UniAcc(:(lp, .(var(x), t)))
    uniacc:=_var_term ∈ (∉L(x, vars_T(t));
                         UniAcc(:=_LPT(x, t, lp))
                        ) UniAcc(:(lp, .(var(x), t)))
    uniacc_var_fun ∈ (UniAcc(:(lp, .(var(x), fun(f, lt))))
                     ) UniAcc(:(lp, .(fun(f, lt), var(x))))
    uniacc_fun_fun ∈ (lt1 ∈ VTerm(n1);
                      lt2 ∈ VTerm(n2);
                      lp ∈ ListPT;
                      ∨(¬(=(f, g)), ¬(=(n1, n2)))
                     ) UniAcc(:(lp, .(fun(f, lt1), fun(g, lt2))))
    uniacc_zip_fun_fun ∈ (f ∈ Fun;
                          UniAcc(++(zip(lt1, lt2), lp))
                         ) UniAcc(:(lp, .(fun(f, lt1), fun(f, lt2))))
```

Figure 4.3: The UniAcc Predicate

function as the ALF function zip (see section C.4.3 for the ALF definition of the function zip) since the former is defined for any two lists while the latter is only defined for two lists of terms of the same length. For this reason, in the last ALF constructor both vectors of terms should be declared with the same length. Finally, notice that in the last introduction rule of the predicate and in its corresponding ALF constructor, we directly use the function symbol f twice instead of using both function symbols f and g and having $f = g$ as a premise of the rule. Similarly, in the second rule we use the variable x twice instead of using the variables x and y and having $x = y$ as a premise of the rule.

Given the definition of the predicate UniAcc, it is possible to show that all lists of pairs of terms satisfy the predicate. We discuss such a proof in appendix B. There, we present the function $\mathsf{allUniAcc}_{\mathrm{LPT}}$ that, given a list of pairs of terms, returns a proof that the list satisfies the predicate UniAcc.

4.5.2 The Unification Algorithm using the UniAcc Predicate

We now describe how we can write the algorithm Unify in Martin-Löf's type theory. This algorithm is the formalisation of the unification algorithm that uses the UniAcc predicate to handle the recursive calls.

As before, the algorithm Unify calls the algorithm unify, but now it has to supply a proof that the input list satisfies the predicate UniAcc.

$$\begin{array}{l}\mathsf{Unify} \in (lp \in \mathsf{ListPT}) \vee (\mathsf{Subst}, \mathsf{Error}) \\ \quad \mathsf{Unify}(lp) \equiv \mathsf{unify}(lp, [], \mathsf{allUniAcc}_{\mathrm{LPT}}(lp))\end{array}$$

The algorithm unify is defined by recursion on the proof that the input list of pairs of terms satisfies the predicate UniAcc. Once we have performed pattern matching over the proof that the input list satisfies the predicate UniAcc, we obtain the following incomplete ALF code with seven equations, one equation for each of the constructors of the predicate UniAcc:

$$\begin{array}{l}\mathsf{unify} \in (lp \in \mathsf{ListPT};\ sb \in \mathsf{Subst};\ \mathsf{UniAcc}(lp)) \vee (\mathsf{Subst}, \mathsf{Error}) \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{[]}) \equiv ?_{\mathrm{unify.0.0.E}} \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathrm{var\_var}}(x, h_1)) \equiv ?_{\mathrm{unify.0.1.E}} \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathrm{var\_term}}(lp_1, h_1, h_2)) \equiv ?_{\mathrm{unify.0.2.E}} \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{:=\mathrm{var\_term}}(h_1, h_2)) \equiv ?_{\mathrm{unify.0.3.E}} \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathrm{var\_fun}}(h_1)) \equiv ?_{\mathrm{unify.0.4.E}} \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv ?_{\mathrm{unify.0.5.E}} \\ \quad \mathsf{unify}(\_, sb, \mathsf{uniacc}_{\mathrm{zip\_fun\_fun}}(f, h_1)) \equiv ?_{\mathrm{unify.0.6.E}}\end{array}$$

Notice that each of these constructors determines the form of the input list, which is shown in ALF by replacing the variable that denotes the input list with the symbol "_". Then, the parameter that represents the input list does not contribute to the understanding of the algorithm and it can be hidden (which is actually done in the presentation of this algorithm in section C.5.3). The first, third and sixth equations correspond to the cases where the algorithm returns a basic result and it is easy to fill them in. In the rest of the equations we have to perform a recursive call, and then we have to supply the new list to

$$\begin{aligned}
&\mathrm{Unify} \in (lp \in \mathrm{ListPT}) \vee(\mathrm{Subst}, \mathrm{Error})\\
&\quad \mathrm{Unify}(lp) \equiv \mathrm{unify}(lp, [], \mathrm{allUniAcc}_{\mathrm{LPT}}(lp))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{unify} \in (lp \in \mathrm{ListPT};\ sb \in \mathrm{Subst};\ \mathrm{UniAcc}(lp)) \vee(\mathrm{Subst}, \mathrm{Error})\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{[]}) \equiv \vee_{\mathrm{L}}(sb)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{var_var}}(x, h_1)) \equiv \mathrm{unify}(lp_1, sb, h_1)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{var_term}}(lp_1, h_1, h_2)) \equiv \vee_{\mathrm{R}}(\mathrm{error})\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{:=\mathrm{var_term}}(h_1, h_2)) \equiv \mathrm{unify}(:=_{\mathrm{LPT}}(x, t, lp_1), :(:=_{\mathrm{S}}(x, t, sb), .(x, t)), h_2)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{var_fun}}(h_1)) \equiv \mathrm{unify}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f, lt))), sb, h_1)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{fun_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv \vee_{\mathrm{R}}(\mathrm{error})\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{zip_fun_fun}}(f, h_1)) \equiv \mathrm{unify}(++(\mathrm{zip}(lt_1, lt_2), lp_1), sb, h_1)
\end{aligned}$$

Figure 4.4: Formalisation of the Unification Algorithm by using the UniAcc Predicate

be unified, the accumulated substitution and a proof that the new list to be unified satisfies the UniAcc predicate. Observe that in each of the recursive equations, this proof is one of the parameters of the constructor that builds a proof that the original list satisfies the predicate UniAcc, that is, it is one of the premises of the corresponding introduction rule. Then, we select the parameter that corresponds to this proof (that is, that the new list to be unified satisfies the predicate UniAcc) and we supply it to the recursive call. As this proof determines the new list to be unified, then it only remains to provide the correct substitution for each of the cases. The following is the ALF code obtained so far:

$$\begin{aligned}
&\mathrm{unify} \in (lp \in \mathrm{ListPT};\ sb \in \mathrm{Subst};\ \mathrm{UniAcc}(lp)) \vee(\mathrm{Subst}, \mathrm{Error})\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{[]}) \equiv \vee_{\mathrm{L}}(sb)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{var_var}}(x, h_1)) \equiv \mathrm{unify}(lp_1, ?_{\mathrm{sb1}}, h_1)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{var_term}}(lp_1, h_1, h_2)) \equiv \vee_{\mathrm{R}}(\mathrm{error})\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{:=\mathrm{var_term}}(h_1, h_2)) \equiv \mathrm{unify}(:=_{\mathrm{LPT}}(x, t, lp_1), ?_{\mathrm{sb2}}, h_2)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{var_fun}}(h_1)) \equiv \mathrm{unify}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f, lt))), ?_{\mathrm{sb3}}, h_1)\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{fun_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv \vee_{\mathrm{R}}(\mathrm{error})\\
&\quad \mathrm{unify}(_, sb, \mathrm{uniacc}_{\mathrm{zip_fun_fun}}(f, h_1)) \equiv \mathrm{unify}(++(\mathrm{zip}(lt_1, lt_2), lp_1), ?_{\mathrm{sb4}}, h_1)
\end{aligned}$$

In figure 4.4, we present the complete formalisation of the type theory version of the unification algorithm that uses the predicate UniAcc to handle the recursive calls.

Observe that the ALF code of this version of the algorithm is short and concise. Notice also that we were able to eliminate all the proofs related to the inequalities of lists of pairs of terms from the code of the formalisation of the algorithm.

4.6 Towards Program Extraction

In this section, we discuss a methodology that extracts a Haskell program from the type theory formalisation of the unification algorithm that uses our special-purpose accessibility predicate to handle the recursive calls.

Although ALF does not support program extraction, in this section we assume that we know how to transform ALF expressions like $:(lp, .(\mathsf{var}(x), t))$, $:=_{\mathsf{LPT}}(x, t, lp)$, $a = b$ or $\in_{\mathsf{L}}(x, \mathsf{vars}_{\mathsf{T}}(t))$ into their corresponding Haskell expressions `((Var x,t):lp)`, `(substLPT x t lp)`, `a == b` or `(x `elem` (varsT t))` respectively, where, `Var`, `substLPT`, `elem` and `varsT` are some of the Haskell constructors and functions already introduced in section 4.1.

In section 4.5.1, to make the definition of the predicate UniAcc shorter, in the second introduction rule of the predicate we used the variable x twice instead of using the variables x and y and adding $x = y$ to the premises of the rule. Similarly, we have also simplified the last introduction rule by using only one function symbol and by defining both vectors with the same length. If we extract a program from the current version of the algorithm unify, we would obtain a Haskell program with expressions like `((Var x,Var x):lp)` as part of the left hand side of the equations of the Haskell program. As Haskell does not allow this kind of expression, we should modify our special-purpose accessibility predicate in order to avoid having the same variable occurring more than once in the type of a introduction rule of the predicate UniAcc. Once we have modified the definition of our special-purpose accessibility predicate, we write the type theory formalisation of the unification algorithm that uses this modified predicate to handle the recursive calls. For this, we follow the method described in the previous section. Observe that, since the vectors in the last introduction rule of the predicate are not longer declared with the same length, we need to define a new zip function in ALF. We present the function zip2, the predicate UniAcc2 and the algorithm unify2 (which are the new versions of zip, UniAcc and unify respectively) in figure 4.5. Observe that the definition of the ALF function zip2 is very similar to the definition of the Haskell function `zip`. The difference is that the Haskell function `zip` is defined for any two lists and our ALF function zip2 is defined just for two lists of terms. Notice that only the second and last constructor of the predicate have changed and also notice that the changes do not affect the definition of the unification algorithm. Finally, observe that, once again, we have hidden the declarations of some of the parameters in the definition of the predicate.

As before (see section 4.5.1), each of the introduction rules of the modified special-purpose accessibility predicate has one of the following two patterns:

$$\frac{c_1 \ \cdots \ c_n}{\mathsf{UniAcc2}(lp)} \qquad\qquad \frac{c_1 \ \cdots \ c_n \qquad \mathsf{UniAcc2}(lp')}{\mathsf{UniAcc2}(lp)}$$

where c_i, for $1 \leqslant i \leqslant n$, are the extra conditions that should be satisfied in order to produce a result or to perform a recursive call in the algorithm, lp' is the list on which the algorithm performs the recursion and lp is the input list. Moreover, the introduction rules for the cases where the unification algorithm produces a

zip2 ∈ (lt_1 ∈ VTerm(n_1); lt_2 ∈ VTerm(n_2)) ListPT
  zip2($[]_v$, lt_2) ≡ []
  zip2($:_v(lt'_1, t_1)$, $[]_v$) ≡ []
  zip2($:_v(lt'_1, t_1)$, $:_v(lt'_2, t_2)$) ≡ :(zip2(lt'_1, lt'_2), .(t_1, t_2))

UniAcc2 ∈ (lp ∈ ListPT) **Set**
  $uniacc2_{[]}$ ∈ UniAcc2([])
  $uniacc2_{var\_var}$ ∈ (=(x, y);
    UniAcc2(lp)
) UniAcc2(:(lp, .(var(x), var(y))))
  $uniacc2_{var\_term}$ ∈ (lp ∈ ListPT;
    $\in_L(x, vars_T(t))$;
    ¬(=(var(x), t))
) UniAcc2(:(lp, .(var(x), t)))
  $uniacc2_{:=var\_term}$ ∈ ($\notin_L(x, vars_T(t))$;
    UniAcc2($:=_{LPT}(x, t, lp)$)
) UniAcc2(:(lp, .(var(x), t)))
  $uniacc2_{var\_fun}$ ∈ (UniAcc2(:(lp, .(var(x), fun(f, lt))))
) UniAcc2(:(lp, .(fun(f, lt), var(x))))
  $uniacc2_{fun\_fun}$ ∈ (lt_1 ∈ VTerm(n_1);
    lt_2 ∈ VTerm(n_2);
    lp ∈ ListPT;
    ∨(¬(=(f, g)), ¬(=(n_1, n_2)))
) UniAcc2(:(lp, .(fun(f, lt_1), fun(g, lt_2))))
  $uniacc2_{zip\_fun\_fun}$ ∈ (=(n_1, n_2);
    =(f, g);
    UniAcc2(++(zip2(lt_1, lt_2), lp))
) UniAcc2(:(lp, .(fun(f, lt_1), fun(g, lt_2))))

unify2 ∈ (lp ∈ ListPT; sb ∈ Subst; UniAcc2(lp)) ∨(Subst, Error)
  unify2(_, sb, $uniacc2_{[]}$) ≡ $\vee_L(sb)$
  unify2(_, sb, $uniacc2_{var\_var}(c_1, h)$) ≡ unify2($lp_1$, sb, h)
  unify2(_, sb, $uniacc2_{var\_term}(lp_1, c_1, c_2)$) ≡ $\vee_R$(error)
  unify2(_, sb, $uniacc2_{:=var\_term}(c_1, h)$) ≡ unify2($:=_{LPT}(x, t, lp_1)$, $:(:=_S(x, t, sb), .(x, t))$, h)
  unify2(_, sb, $uniacc2_{var\_fun}(h)$) ≡ unify2(:(lp_1, .(var(x), fun(f, lt))), sb, h)
  unify2(_, sb, $uniacc2_{fun\_fun}(lt_1, lt_2, lp_1, c_1)$) ≡ $\vee_R$(error)
  unify2(_, sb, $uniacc2_{zip\_fun\_fun}(c_1, c_2, h)$) ≡ unify2(++(zip2($lt_1$, lt_2), lp_1), sb, h)

Figure 4.5: Definitions of the function zip2, the predicate UniAcc2 and the algorithm unify2

basic result (either the value error or a substitution) follow the pattern of the left rule above, while the introduction rules for the cases where the unification algorithm performs a recursive call follow the pattern of the right rule above.

Then, we can also distinguish two kinds of equations in the ALF code of the algorithm unify2: the equations where the algorithm produces a basic result, which have the form:

$$\mathsf{unify2}(\_, sb, \mathsf{uniacc2}_{\mathsf{name}}(v_1, \ldots, v_m, c_1, \ldots, c_n)) = \mathsf{basic\_result}$$

and the equations where the algorithm performs a recursive call, which have the form:

$$\mathsf{unify2}(\_, sb, \mathsf{uniacc2}_{\mathsf{name}}(v_1, \ldots, v_m, c_1, \ldots, c_n, h)) = \mathsf{unify2}(lp', sb', h)$$

where v_i, for $1 \leqslant i \leqslant m$, are the declarations of the variables that are used in the corresponding introduction rule of the predicate (which are usually hidden, so in many cases one cannot see them), sb is the original accumulated substitution, sb' is the new accumulated substitution, lp' is the list on which we perform the recursive call and h is a proof that lp' satisfies the predicate UniAcc.

Now, we describe how to obtain the Haskell equation that corresponds to each of the two type theory equations presented above.

Given the type theory equation of the algorithm unify2 that produces a result, we know there exists a list of pairs of terms lp such that:

$$\mathsf{uniacc2}_{\mathsf{name}}(v_1, \ldots, v_m, c_1, \ldots, c_n) \in \mathsf{UniAcc2}(lp)$$

Notice that lp is the input list of the algorithm. Moreover, the variables used to form the list lp are drawn from $v_1, \ldots, v_m$.

Let `sb`, `lp`, `basic_result` and `ci` be the Haskell versions of sb, lp, basic_result and c_i respectively, for $1 \leqslant i \leqslant n$. Then, the Haskell equation corresponding to the equation of the algorithm unify2 that produces a result is the following:

```
unify2 lp sb
    | c1 && ... && cn = basic_result
```

Similarly, given the type theory equation of the algorithm unify2 that performs a recursive call, we know that there exist two lists of pairs of terms lp' and lp such that:

$$h \in \mathsf{UniAcc2}(lp')$$
$$\mathsf{uniacc2}_{\mathsf{name}}(v_1, \ldots, v_m, c_1, \ldots, c_n, h) \in \mathsf{UniAcc2}(lp)$$

Notice that lp is the input list and lp' the list on which we perform the recursion. As before, the variables used to form both the list lp' and the list lp are drawn from $v_1, \ldots, v_m$. Moreover, the variables used to form lp' are included in the ones used to form lp.

Let `sb`, `sb'`, `lp`, `lp'` and `ci` be the Haskell versions of sb, sb', lp, lp' and c_i respectively, for $1 \leqslant i \leqslant n$. Then, the Haskell equation corresponding to the equation of the algorithm unify2 that performs a recursive call is the following:

```
unify2 lp sb
    | c1 && ... && cn = unify2 lp' sb'
```

Following this explanation, the Haskell algorithm that would be extracted from the algorithm unify2 would be the following one:

```
unify :: ListPT -> Subst -> Either Subst Error

unify2 []                 sb = Right sb
unify2 ((Var x,Var y):lp) sb
       | x == y              = unify2 lp sb
unify2 ((Var x,t):lp)     sb
       | x `elem` (varsT t) && (Var x) /= t = Left Error
unify2 ((Var x,t):lp)     sb
       | not(x `elem` (varsT t)) = unify2 (substLPT x t lp)
                                          ((x,t):substS x t sb)
unify2 ((Fun f lt,Var x):lp)      sb
                        = unify2 ((Var x, Fun f lt):lp) sb
unify2 ((Fun f lt1,Fun g lt2):lp) sb
       | f /= g || length lt1 /= length lt2 = Left Error
unify2 ((Fun f lt1,Fun g lt2):lp) sb
       | f == g && length lt1 == length lt2
                        = unify2 ((zip lt1 lt2)++lp) sb
```

Notice that the algorithm that results from the program extraction is very similar to the Haskell algorithm we presented in section 4.1 and that we used for constructing the predicate UniAcc (and the predicate UniAcc2). This is not a surprise to us, since we can actually think of the process that given a Haskell version of the unification algorithm constructs its type theory version, and the process that extracts a Haskell program from the type theory version of the algorithm as being the inverse of each other. The fact that the type of the result of the algorithm given above is `Either Subst Error` instead of `Maybe Subst` (as in the Haskell program of section 4.1) has to do with the decision to formalise the type `Maybe Subst` as $\vee(\mathsf{Subst}, \mathsf{Error})$ in type theory. In addition, notice that the equations in the algorithm we present above are exhaustive and mutually disjoint. This comes from the fact that the introduction rules of the special-purpose accessibility predicate are also exhaustive and mutually disjoint. In addition, as each equation in the above algorithm considers one and only one possible case of the input list of pairs of terms, they can be given in any order. Finally, we want to add that the transformation that takes the predicate UniAcc into the predicate UniAcc2 can be done completely automatically. Then, when a variable x occurs more than once in the type of an introduction rule, we should generate new variables to replace the repeated occurrences of the variable x and we should add conditions stating that the new generated variables are equal to x. Moreover, we could have performed the generation of new variables and the addition of the new constraints embedded in the program extraction process, and then we could have just presented the program extraction methodology from

the predicate UniAcc. However, this would have only made the real process of program extraction more complicated.

To conclude, we believe that this methodology for program extraction is easy to program, and then it can be added as part of a future program extraction module for ALF.

Chapter 5

More about Substitutions

We introduced the ALF formalisation of substitutions in section 4.3. Here, we introduce a few more definitions and some properties of substitutions which will be used in the following two chapters to prove the partial correctness of the algorithm Unify and the integrated approach to the unification algorithm.

5.1 Application of Substitutions

We define two ways of applying a substitution sb to a term t: by recursion on the term and by recursion on the substitution. Both definitions are useful when proving properties that involve the result of applying a substitution to a term. The definition that is more convenient in each case depends on the property we want to prove.

The ALF definition of the application of a substitution to a term that is defined by recursion on the term is as follows:

$\text{appP}_\text{T} \in (sb \in \text{Subst};\ t \in \text{Term})\ \text{Term}$
$\quad \text{appP}_\text{T}([], \text{var}(x)) \equiv \text{var}(x)$
$\quad \text{appP}_\text{T}(:(sb_1, .(x_1, t)), \text{var}(x)) \equiv \text{Var}_\text{to}\text{A}(t, \text{appP}_\text{T}(sb_1, \text{var}(x)), \text{Var}_\text{dec}(x_1, x))$
$\quad \text{appP}_\text{T}(sb, \text{fun}(f, lt)) \equiv \text{fun}(f, \text{appP}_\text{VT}(sb, lt))$
$\text{appP}_\text{VT} \in (sb \in \text{Subst};\ lt \in \text{VTerm}(n))\ \text{VTerm}(n)$
$\quad \text{appP}_\text{VT}(sb, []_\text{v}) \equiv []_\text{v}$
$\quad \text{appP}_\text{VT}(sb, :_\text{v}(lt', t)) \equiv :_\text{v}(\text{appP}_\text{VT}(sb, lt'), \text{appP}_\text{T}(sb, t))$

where the function $\text{Var}_\text{to}\text{A}$ was already introduced at the beginning of section 4.3.

Notice that due to the way terms are defined, we need two mutually recursive functions to define this application. Notice also that this definition corresponds to what is usually referred as *parallel application*. In the parallel application, we substitute, at the same time, all the terms that occur in the right hand side of the pairs of a substitution sb for their associated variables in the term t.

The ALF definition of the application of a substitution to a term that is defined by recursion on the substitution is as follows:

$\text{appS}_T \in (sb \in \text{Subst};\ t \in \text{Term})\ \text{Term}$
$\text{appS}_T([], t) \equiv t$
$\text{appS}_T(:(sb_1, .(x, t_1)), t) \equiv \text{appS}_T(sb_1, :=_T(x, t_1, t))$

Notice that this definition corresponds to what is usually referred as *sequential application*. In the sequential application, we substitute, one at a time, all the terms that occur in the right hand side of the pairs of a substitution sb for their associated variables in the term t, until there are no more variables in the domain of the substitution, that is, the substitution is empty.

Even though the sequential application can be defined using just one function, it is useful to define the corresponding sequential application for vectors of terms. Then, we have that:

$\text{appS}_{VT} \in (sb \in \text{Subst};\ lt \in \text{VTerm}(n))\ \text{VTerm}(n)$
$\text{appS}_{VT}(sb, []_v) \equiv []_v$
$\text{appS}_{VT}(sb, :_v(lt', t)) \equiv :_v(\text{appS}_{VT}(sb, lt'), \text{appS}_T(sb, t))$

For a given substitution and a given term, the application of the substitution to the term might result in different terms, depending on whether one follows the definition of the parallel application or the definition of the sequential application. As an example, given the substitution $[(x,\ g(y)),(y,\ h(z))]$ and the term $f(x)$, the result of the parallel application is the term $f(g(y))$, while the result of the sequential application is the term $f(g(h(z)))$. However, in section 5.4 we show that both applications give the same result for idempotent substitutions.

As we already mention, both the parallel and the sequential application are useful when proving properties that involve the result of applying a substitution to a term. Besides, the result of both application is the same for idempotent substitution. Since this is actually our case, as it will be shown in section 6.3, it does not matter which of the definitions we choose when proving properties.

In what follows, we use $sb(t)$ or we refer to "the application of sb to t" to denote both the parallel and the sequential application of a substitution sb to a term t. Given a vector of terms lt, an analogous explanation holds for $sb(lt)$.

5.2 Idempotent Substitutions

Following the standard definition of idempotence, a substitution sb is *idempotent* if for every term t it holds that $sb(sb(t)) = sb(t)$. The ALF definition of idempotence is the following:

$\text{Idempotent} \in (sb \in \text{Subst})\ \textbf{Set}$
$\text{Idempotent} \equiv [sb]\forall(\text{Term}, [t]=(\text{appP}_T(sb, \text{appP}_T(sb, t)), \text{appP}_T(sb, t)))$

Since this definition does not contain any concrete information that helps us in understanding when the equality holds, this definition is sometimes not very useful if we intend to use the fact that a certain substitution is idempotent to prove other properties of the substitution. Therefore, we want to have an inductive definition of idempotence.

To understand the definition of idempotence we need to understand under which conditions the equality $sb(sb(t)) = sb(t)$ holds. Applying a substitution

to a term results in the same term only when the domain of the substitution and the set of variables in the term are disjoint. In our particular case, if the domain and the range of the substitution sb are disjoint, then the domain of sb and the set of variables in $sb(t)$ are disjoint. Thus, we give the following inductive definition of idempotence:

```
Idem ∈ (sb ∈ Subst) Set
    []idem ∈ Idem([])
    :idem ∈ (Idem(sb);
             Disjoint(varsT(t), dom(:(sb, .(x, t))));
             ∉L(x, varsS(sb))
            ) Idem(:(sb, .(x, t)))
```

Notice that this definition provides more information than just the fact that the domain and the range of a substitution are disjoint. It also states that all the variables in the domain of a substitution that satisfies the predicate Idem are different from each other.

In what follows, we usually refer to "an idempotent substitution" when we actually mean "a substitution that satisfies the predicate Idem".

5.3 Most General Unifier

In this section, we present several definitions that involve the notion of unifier, concluding with the definition of the notion of most general unifier.

We now define when a substitution sb unifies a list of pairs of terms lp. As explained before, sb unifies the list lp if, for all pair (t_1, t_2) in lp, it holds that $sb(t_1) = sb(t_2)$. Hence, we give the following inductive definition in ALF:

```
unifiesLPT ∈ (sb ∈ Subst; lp ∈ ListPT) Set
    unifiesLPT_[] ∈ (sb ∈ Subst) unifiesLPT(sb, [])
    unifiesLPT_: ∈ (unifiesLPT(sb, lp');
                    =(appPT(sb, t1), appPT(sb, t2))
                   ) unifiesLPT(sb, :(lp', .(t1, t2)))
```

In a similar way, we define when a substitution sb_1 unifies a substitution sb_2.

```
unifiesS ∈ (sb1, sb2 ∈ Subst) Set
    unifiesS_[] ∈ (sb ∈ Subst) unifiesS(sb, [])
    unifiesS_: ∈ (unifiesS(sb, sb1);
                  =(appPT(sb, var(x)), appPT(sb, t))
                 ) unifiesS(sb, :(sb1, .(x, t)))
```

We want to prove that if the substitution sb is the result of unifying the list of pairs of terms lp, then lp and sb are *equivalent* in the sense that they have the same set of unifiers when we consider both lp and sb as set of equations. Then, we need to have a notion of *equivalence* between lists of pairs of terms and substitutions. As the algorithm Unify is defined in terms of the algorithm unify and the latter takes also an accumulated substitution as input, we need to give a more general notion of equivalence in order to be able to prove the desired property. Given a list of pairs of terms lp and two substitutions sb and

sb', we define that the pair (lp, sb) is equivalent to the substitution sb' if every substitution sb_1 that unifies both lp and sb also unifies sb', and vice versa.

```
≡LpSbLpSb ∈ (lp ∈ ListPT; sb, sb' ∈ Subst) Set
    ≡LpSbLpSb ≡
        [lp, sb, sb']
            ∀(Subst, [sb1]⇔(∧(unifiesLPT(sb1, lp), unifiesS(sb1, sb)), unifiesS(sb1, sb')))
```

We now define the desired notion of equivalence between a list of pairs of terms and a substitution as a special case of the previous notion.

```
≡LpSb ∈ (lp ∈ ListPT; sb ∈ Subst) Set
    ≡LpSb ≡ [lp, sb]≡LpSbLpSb(lp, [], sb)
```

Finally, we define the notion of most general unifier. We defined before, a substitution sb is a most general unifier of a list of pairs of terms lp if sb is the most general substitution that unifies lp. In other words, sb is a most general unifier of lp if sb unifies lp and, for any other substitution sb' that also unifies lp, sb is at least as general as sb'. The relation "at least as general as" on substitutions is defined as follows: sb is at least as general as sb' if there exists a substitution sb_1 such that $sb'(t) = sb_1(sb(t))$, for all terms t. The Alf formalisation of this relation is the following:

```
≤Sb ∈ (sb, sb' ∈ Subst) Set
    ≤Sb ≡ [sb, sb']∃(Subst, [sb1]∀(Term, [t]=(appPT(sb', t), appPT(sb1, appPT(sb, t)))))
```

With this definition, we write $sb \leq_{\mathrm{Sb}} sb'$ whenever sb is at least as general as sb'.

We express the notion of most general unifier in the following ALF definition:

```
mgu ∈ (sb ∈ Subst; lp ∈ ListPT) Set
    mgu ≡
        [sb, lp]∧(unifiesLPT(sb, lp), ∀(Subst, [sb']⇒(unifiesLPT(sb', lp), ≤Sb(sb, sb'))))
```

5.4 Some Properties involving Substitutions

In this section, we present some interesting properties involving substitutions. See section C.4.6 for the complete ALF proofs of these properties. Although the results we show here are used in the following two chapters, the reader may skip the technical details in the proofs of these results.

The first property we present here states that, given a variable x, two terms t and t' and a substitution sb, if x and t have the same image under sb, then the terms t' and $t'\,[x{:=}t]$ also have the same image under sb. Because of the way terms are defined, we also need the corresponding property for vectors of terms.

```
=:=appP_T ∈ (t' ∈ Term;
              =(appPT(sb, var(x)), appPT(sb, t))
             ) =(appPT(sb, t'), appPT(sb, :=T(x, t, t')))
=:=appP_VT ∈ (lt ∈ VTerm(n);
               =(appPT(sb, var(x)), appPT(sb, t))
              ) =(appPVT(sb, lt), appPVT(sb, :=VT(x, t, lt)))
```

The proofs are made by recursion on the term t' and the vector lt respectively.

As expected, if the set of variables in a term and the domain of a substitution are disjoint, applying the substitution to the term has no effect.

$$=_{T\text{disj\_vars}} \in (sb \in \text{Subst};\ \text{Disjoint}(\text{vars}_T(t), \text{dom}(sb))) = (t, \text{appS}_T(sb, t))$$

This property is proven by recursion on the substitution sb.

The next property establishes that if sb is an idempotent substitution, then the set of variables in the term $sb(t)$ and the domain of sb are disjoint.

$$\text{disj}_{\text{vars\_appS}} \in (t \in \text{Term};\ \text{Idem}(sb))\ \text{Disjoint}(\text{vars}_T(\text{appS}_T(sb, t)), \text{dom}(sb))$$

The proof is made by recursion on the proof that sb is idempotent.

An important property already mentioned before is that, for idempotent substitutions, the parallel and sequential applications produce the same result. Once more, due to the way terms are defined, we need the corresponding property for vectors of terms.

$$=_T\text{appP-S} \in (sb \in \text{Subst};\ t \in \text{Term};\ \text{Idem}(sb)) = (\text{appP}_T(sb, t), \text{appS}_T(sb, t))$$
$$=_{VT}\text{appP-S} \in (lt \in \text{VTerm}(n);\ \text{Idem}(sb)) = (\text{appP}_{VT}(sb, lt), \text{appS}_{VT}(sb, lt))$$

The proofs are made by recursion on the term t and the vector lt respectively. When the term is a variable term, we also consider cases on the substitution sb.

Now, we present the property that establishes that if a substitution satisfies the predicate Idem, then the substitution is idempotent according to the standard definition.

$$\text{idem}_{to}\text{idempotent} \in (\text{Idem}(sb))\ \text{Idempotent}(sb)$$

This proof uses the previous three properties presented here.

Finally, we present the proof that states that if a variable (term) x is included in the set of variables in a term t with $x \neq t$, then there exists no substitution that unifies x and t.

The usual way to prove this property consists in defining an inductive relation "is a proper subterm of", showing that x is a proper subterm of t, and then proving that this relation is preserved under substitution application. Hence, for no substitution sb we have that $sb(x) = sb(t)$ since $sb(x)$ is a proper subterm of $sb(t)$.

The approach we use to prove this property is a different one. We prove an auxiliary lemma that establishes that, given a substitution sb, it is absurd to have that $x \in \text{vars}_T(t)$ with $x \neq t$ and also that $sb(x) = sb(t)$. Then, proving that no substitution can unify x and t is trivial from this auxiliary lemma.

In what follows, we make use of some lemmas about inequality of natural numbers[1]:

$$\leq_{to}<_{sL} \in (\leq(n, m)) <(n, s(m))$$
$$<\wedge=_{to}\bot \in (<(n, m);\ =(n, m))\ \bot$$
$$\leq_{to}\leq_{+R} \in (p \in \text{N};\ \leq(n, m))\ \leq(n, +(m, p))$$

We now present the auxiliary lemma in ALF:

[1] See section C.3.4 for the complete ALF proofs of these lemmas.

$$\in_{\wedge\neq\wedge}\mathrm{unify}_{\mathrm{to}}\bot \in (t \in \mathrm{Term};\ \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t));\ \neg(=(\mathrm{var}(x), t));\ =(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x)), \mathrm{appP}_{\mathrm{T}}(sb, t)))\ \bot$$
$$\in_{\wedge\neq\wedge}\mathrm{unify}_{\mathrm{to}}\bot(\mathrm{var}(x_1), \in_{\mathrm{hd}}(\_, \_), \Rightarrow_{\mathrm{I}}(f_1), h_2) \equiv f_1(\mathrm{refl}(\mathrm{var}(x_1)))$$
$$\in_{\wedge\neq\wedge}\mathrm{unify}_{\mathrm{to}}\bot(\mathrm{var}(x_1), \in_{\mathrm{tl}}(\_, \_, h_3), h_1, h_2) \equiv \textbf{case}\ h_3 \in_{\mathrm{L}}(x, [])\ \textbf{of}\ \textbf{end}$$
$$\in_{\wedge\neq\wedge}\mathrm{unify}_{\mathrm{to}}\bot(\mathrm{fun}(f, lt), h, h_1, h_2) \equiv <\wedge=_{\mathrm{to}}\bot(\leq_{\mathrm{to}}<_{\mathrm{sL}}(\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, lt, h)), =_{\mathrm{congl}}(\#\mathrm{funs}_{\mathrm{T}}, h_2))$$

The lemma is proven by first performing pattern matching on the term t. When t is a variable, we study cases on the proof that $x \in \mathrm{vars}_{\mathrm{T}}(t)$. Clearly, t cannot be the variable x because $x \neq t$ (first equation) nor a variable different from x because $x \in \mathrm{vars}_{\mathrm{T}}(t)$ (second equation). Hence, t has to be a function application of the form $f(lt)$, and then we have that $x \in \mathrm{vars}_{\mathrm{VT}}(lt)$. Here, we know that $\#\mathrm{funs}_{\mathrm{T}}(f(lt)) = \#\mathrm{funs}_{\mathrm{VT}}(lt) + 1$, by definition of the function $\#\mathrm{funs}_{\mathrm{T}}$. Now, the lemma $\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}$ (which is explained below) gives us a proof that $\#\mathrm{funs}_{\mathrm{T}}(sb(x)) \leq \#\mathrm{funs}_{\mathrm{VT}}(sb(lt))$ which, by lemma $\leq_{\mathrm{to}}<_{\mathrm{sL}}$, gives us a proof that $\#\mathrm{funs}_{\mathrm{T}}(sb(x)) < \#\mathrm{funs}_{\mathrm{T}}(sb(f(lt)))$. On the other hand, as $sb(x) = sb(f(lt))$, we obtain that $\#\mathrm{funs}_{\mathrm{T}}(sb(x)) = \#\mathrm{funs}_{\mathrm{T}}(sb(f(lt)))$ which clearly contradicts the previous result.

The lemmas $\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}$ and $\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}$ are defined in a mutually recursive way as follows:

$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}} \in (sb \in \mathrm{Subst};\ t \in \mathrm{Term};\ \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t)))\leq(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x))), \#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, t)))$$
$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, \mathrm{var}(x_1), \in_{\mathrm{hd}}(\_, \_)) \equiv \vee_{\mathrm{R}}(\mathrm{refl}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x_1)))))$$
$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, \mathrm{var}(x_1), \in_{\mathrm{tl}}(\_, \_, h_1)) \equiv \textbf{case}\ h_1 \in_{\mathrm{L}}(x, [])\ \textbf{of}\ \textbf{end}$$
$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, \mathrm{fun}(f, lt), h) \equiv \leq_{\mathrm{to}}\leq_{+\mathrm{R}}(\mathrm{s}(0), \in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, lt, h))$$
$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}} \in (sb \in \mathrm{Subst};\ lt \in \mathrm{VTerm}(n);\ \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}(lt)))\leq(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x))), \#\mathrm{funs}_{\mathrm{VT}}(\mathrm{appP}_{\mathrm{VT}}(sb, lt)))$$
$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, []_{\mathrm{v}}, h) \equiv \textbf{case}\ h \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}([]_{\mathrm{v}}))\ \textbf{of}\ \textbf{end}$$
$$\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, :_{\mathrm{v}}(lt', t'), h) \equiv$$
$$\textbf{case}\ \in+\!\!+_{\mathrm{to}}\in\vee(\mathrm{vars}_{\mathrm{T}}(t'), h) \in \vee(\in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}(lt')), \in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t')))\ \textbf{of}$$
$$\vee_{\mathrm{L}}(h_1) \Rightarrow \leq_{\mathrm{to}}\leq_{+\mathrm{R}}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, t')), \in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}(sb, lt', h_1))$$
$$\vee_{\mathrm{R}}(h_2) \Rightarrow$$
$$=_{\mathrm{subst1}}(+_{\mathrm{comm}}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{appP}_{\mathrm{T}}(sb, t')), \#\mathrm{funs}_{\mathrm{VT}}(\mathrm{appP}_{\mathrm{VT}}(sb, lt'))),$$
$$\leq_{\mathrm{to}}\leq_{+\mathrm{R}}(\#\mathrm{funs}_{\mathrm{VT}}(\mathrm{appP}_{\mathrm{VT}}(sb, lt')), \in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}(sb, t', h_2)))$$
$$\textbf{end}$$

The lemmas are proven by recursion on the term t and the vector of terms lt respectively.

When t is a variable, we study cases on the proof that $x \in \mathrm{vars}_{\mathrm{T}}(t)$. If t is the variable x, then the result is trivial (first equation of lemma $\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}$). On the other hand, t cannot be a variable different from x because this contradicts the fact that $x \in \mathrm{vars}_{\mathrm{T}}(t)$ (second equation of lemma $\in_{\mathrm{to}}\leq_{\#\mathrm{funsT}}$). When t is a function application of the form $f(lt)$, we know that $x \in \mathrm{vars}_{\mathrm{VT}}(lt)$. By definition of the function $\#\mathrm{funs}_{\mathrm{T}}$, we have that $\#\mathrm{funs}_{\mathrm{T}}(f(lt)) = \#\mathrm{funs}_{\mathrm{VT}}(lt) + 1$. Here, by lemma $\in_{\mathrm{to}}\leq_{\#\mathrm{funsVT}}$, we have a proof that $\#\mathrm{funs}_{\mathrm{T}}(sb(x)) \leq \#\mathrm{funs}_{\mathrm{VT}}(sb(lt))$ which, by lemma

$\leq_{\mathrm{to}} \leq_{+\mathrm{R}}$, gives us a proof that $\#\mathsf{funs}_{\mathrm{T}}(sb(x)) \leq \#\mathsf{funs}_{\mathrm{T}}(sb(f(lt)))$.

The vector of terms lt cannot be empty because this contradicts the fact that $x \in \mathsf{vars}_{\mathrm{VT}}(lt)$. Hence, it should be of the form $(t' : lt')$. Here, by definition of the function $\#\mathsf{funs}_{\mathrm{VT}}$, we have that $\#\mathsf{funs}_{\mathrm{VT}}(lt) = \#\mathsf{funs}_{\mathrm{VT}}(lt') + \#\mathsf{funs}_{\mathrm{T}}(t')$. Now, we use the lemma $\in\!\!+\!\!+_{\mathrm{to}}\!\in\!\vee$, with the proof h that $x \in \mathsf{vars}_{\mathrm{VT}}(lt)$, to see whether $x \in \mathsf{vars}_{\mathrm{VT}}(lt')$ or $x \in \mathsf{vars}_{\mathrm{T}}(t')$. If $x \in \mathsf{vars}_{\mathrm{VT}}(lt')$ (first equation in the case analysis), by recursion on the vector lt' we have that $\#\mathsf{funs}_{\mathrm{T}}(sb(x)) \leq \#\mathsf{funs}_{\mathrm{VT}}(sb(lt'))$, and then we obtain $\#\mathsf{funs}_{\mathrm{T}}(sb(x)) \leq \#\mathsf{funs}_{\mathrm{VT}}(sb(lt')) + \#\mathsf{funs}_{\mathrm{T}}(sb(t'))$ by lemma $\leq_{\mathrm{to}} \leq_{+\mathrm{R}}$. The case where $x \in \mathsf{vars}_{\mathrm{T}}(t')$ is similar to the previous one. Here, as the order of the summands in the definition of $\#\mathsf{funs}_{\mathrm{VT}}$ is relevant in ALF, we have to use the fact that the addition of natural numbers is commutative.

Chapter 6

Partial Correctness of the Unification Algorithm

Here, we present the partial correctness of the unification algorithm introduced in section 4.5. Then, we prove the following properties:

- The algorithm Unify results in the value error only if there exists no substitution that unifies the input list of pairs of terms.
- If the unification algorithm results in a substitution, then the variables that occur in this substitution are included in the variables that occur in the input list of pairs of terms.
- If the unification algorithm results in a substitution, then this substitution is idempotent.
- If the unification algorithm results in a substitution, then this substitution is a most general unifier of the input list of pairs of terms.

Each of the following four sections describes how to prove one of the above properties. See sections C.6 and C.5.4 for the ALF codes of the formalisation of these properties.

We assume that, by now, the reader is already familiar with the ALF notation and with the way properties are proven in ALF. Therefore, we do not explain the following ALF codes as much as we have done it previously. In addition, as we prove each of the properties in a similar way, only the first two properties are described in a more detailed way.

6.1 About the Result of the Unification Algorithm

In this section we show that if there exists a substitution that unifies the input list of pairs of terms, then the result of the unification algorithm is not the value

error, and if the result of the unification algorithm is the value error, then there exists no substitution that unifies the input list of pairs of terms. The ALF definitions of these two functions are the following:

$$\text{unifies}_{to\neg\neg}\text{error} \in (\exists(\text{Subst}, [sb]\text{unifies}_{LPT}(sb, lp))) \neg(=(\text{Unify}(lp), \vee_R(\text{error})))$$
$$\text{unifies}_{to\neg\neg}\text{error}(h) \equiv \Rightarrow_I([h']\text{error} \wedge \text{unifies}_{to\perp}(h', h))$$
$$\text{error}_{to\neg\neg}\text{unifies} \in (=(\text{Unify}(lp), \vee_R(\text{error}))) \neg(\exists(\text{Subst}, [sb]\text{unifies}_{LPT}(sb, lp)))$$
$$\text{error}_{to\neg\neg}\text{unifies}(h) \equiv \Rightarrow_I(\text{error} \wedge \text{unifies}_{to\perp}(h))$$

To prove these properties we use the following lemma:

$$\text{error} \wedge \text{unifies}_{to\perp} \in (=(\text{Unify}(lp), \vee_R(\text{error})); \exists(\text{Subst}, [sb]\text{unifies}_{LPT}(sb, lp))) \perp$$
$$\text{error} \wedge \text{unifies}_{to\perp}(h, h_1) \equiv \text{unifies} \wedge \text{error}_{to\perp}(\text{allUniAcc}_{LPT}(lp), h_1, h)$$

which, in turn, uses an auxiliary lemma over the algorithm unify. This auxiliary lemma takes, as an extra parameter, a proof that the input list satisfies the predicate UniAcc. We prove this auxiliary lemma by recursion on this extra parameter.

$$\begin{aligned}
&\text{unifies} \wedge \text{error}_{to\perp} \in (p \in \text{UniAcc}(lp);\\
&\qquad \exists(\text{Subst}, [sb']\text{unifies}_{LPT}(sb', lp));\\
&\qquad =(\text{unify}(sb, p), \vee_R(\text{error}))\\
&\qquad)\perp\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{[]}, h, h_1) \equiv \textbf{case } h_1 \in =(\text{unify}(sb, \text{uniacc}_{[]}), \vee_R(\text{error})) \textbf{ of}\\
&\qquad \textbf{end}\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{var_var}(x, h_2), \exists_I(sb_1, h), h_1) \equiv\\
&\quad \text{unifies} \wedge \text{error}_{to\perp}(h_2, \exists_I(sb_1, \text{unifies}_{LPT_red}(h)), h_1)\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{var_term}(lp_1, h_2, h_3), \exists_I(sb_1, h), h_1) \equiv \text{unifies}_{LPT} \wedge \in_{to\perp}(h, h_2, h_3)\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{:=var_term}(h_2, h_3), \exists_I(sb_1, h), h_1) \equiv\\
&\quad \text{unifies} \wedge \text{error}_{to\perp}(h_3, \exists_I(sb_1, \text{unifies}_{LPT_:=R}(h)), h_1)\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{var_fun}(h_2), \exists_I(sb_1, h), h_1) \equiv\\
&\quad \text{unifies} \wedge \text{error}_{to\perp}(h_2, \exists_I(sb_1, \text{unifies}_{LPT_fv_{to}vf}(h)), h_1)\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{fun_fun}(lt_1, lt_2, lp_1, \vee_L(\Rightarrow_I(f_1))), \exists_I(sb_1, h), h_1) \equiv f_1(\text{unifies}_{LPTto=f}(h))\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{fun_fun}(lt_1, lt_2, lp_1, \vee_R(\Rightarrow_I(f_1))), \exists_I(sb_1, h), h_1) \equiv f_1(\text{unifies}_{LPTto=arity}(h))\\
&\text{unifies} \wedge \text{error}_{to\perp}(\text{uniacc}_{zip_fun_fun}(f, h_2), \exists_I(sb_1, h), h_1) \equiv\\
&\quad \text{unifies} \wedge \text{error}_{to\perp}(h_2, \exists_I(sb_1, \text{unifies}_{LPT_fun_{to}zip}(h)), h_1)
\end{aligned}$$

In the first equation, h_1 is a proof that the result of the algorithm unify is the value error which contradicts the fact that when the input list is empty the result of unify is the accumulated substitution.

In the second equation, h is a proof that sb_1 unifies the input list, which has the form $(x, x) : lp'$. Now, the result of unifying the list $(x, x) : lp'$ is, by hypothesis, the value error and, by definition of the algorithm unify, equal to the result of unifying the list lp'. Hence, we have that the result of unifying the list lp' is equal to the value error. Then, by recursion on the proof that the list lp' satisfies the predicate UniAcc (that is, the parameter h_2), we obtain a contradiction. To the recursive call, we have to supply a proof that there exists a substitution that unifies the list lp'. The lemma unify_{LPT_red} takes the proof that sb_1 unifies the input list (that is, it takes the argument h) and gives us a proof that sb_1 also unifies the list lp' (see section C.4.4 for the ALF proof of this lemma).

The third equation considers the case where the input list is of the form

$(x, t) : lp'$, with $x \in t$ and $x \neq t$. As the substitution sb_1 unifies the input list, then we know that $sb_1(x) = sb_1(t)$. The lemma $\mathsf{unifies}_{\mathrm{LPT}} \wedge \in_{\mathrm{to}} \bot$ (see section C.4.4 for its ALF proof) uses the lemma $\in_{\wedge} \neq_{\wedge} \mathsf{unifies}_{\mathrm{to}} \bot$ (presented in section 5.4) to show that this case leads to a contradiction.

The following two equations are similar to the second one.

The next equation considers the case where the input list of pairs of terms has the form $(f(lt_1), g(lt_2)) : lp'$ and we have a proof that $f \neq g$. Now, as the substitution sb_1 unifies the input list, we have that $sb_1(f(lt_1)) = sb_1(g(lt_2))$. Clearly, this can only hold if $f = g$ which contradicts the fact that $f \neq g$. The lemma $\mathsf{unifies}_{\mathrm{LPTto}=\mathrm{f}}$ takes the argument h (that is, the proof that sb_1 unifies the input list), and returns a proof that the function symbols are equal (see section C.4.4 for the ALF formalisation of the proof of this lemma).

The following equation is similar to the previous one. The last equation is similar to the second one.

6.2 Variables Property

To prove that if the algorithm Unify results in a substitution, then the variables that occur in this substitution are included in the variables that occur in the input list of pairs of terms, we use a similar technique as in the previous section. That is, we prove an auxiliary lemma over the algorithm unify that takes, among other parameters, a proof that the input list satisfies the predicate UniAcc.

$$\mathsf{vars}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb))) \subseteq (\mathsf{vars}_{\mathrm{S}}(sb), \mathsf{vars}_{\mathrm{LPT}}(lp))$$
$$\mathsf{vars}_{\mathrm{prop}}(h) \equiv \mathsf{vars}_{\mathrm{lemma}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), h, \subseteq_{\mathrm{refl}}(\mathsf{vars}_{\mathrm{LPT}}(lp)), \subseteq_{[]}(\mathsf{vars}_{\mathrm{LPT}}(lp)))$$

The auxiliary lemma states that if the input list satisfies the predicate UniAcc, if the algorithm unify results in a substitution sb', and if both the variables that occur in the input list of pairs of terms and in the accumulated substitution are included in a list of variables l, then the variables that occur in the substitution sb' are also included in the list l. When we call this lemma from the proposition $\mathsf{vars}_{\mathrm{prop}}$, we use the list $\mathsf{vars}_{\mathrm{LPT}}(lp)$ as the list of variables l. Hence, we need proofs that $\mathsf{vars}_{\mathrm{LPT}}(lp) \subseteq \mathsf{vars}_{\mathrm{LPT}}(lp)$ and $[\,] \subseteq \mathsf{vars}_{\mathrm{LPT}}(lp)$, since $[\,]$ is the initial accumulated substitution.

To prove the auxiliary lemma, we use several general lemmas about list inclusion whose proofs can be found in section C.3.2. The proof of the auxiliary lemma also uses a few special-purpose lemmas about list inclusion whose proofs can be found in section C.4.4.

Below we show the proof of the auxiliary lemma, which is done by recursion on the proof that the input list satisfies the predicate UniAcc.

$\mathsf{vars}_{\mathsf{lemma}} \in (p \in \mathsf{UniAcc}(lp);$
$=(\mathsf{unify}(sb, p), \vee_{\mathsf{L}}(sb'));$
$\subseteq(\mathsf{vars}_{\mathsf{LPT}}(lp), l);$
$\subseteq(\mathsf{vars}_{\mathsf{S}}(sb), l)$
$) \subseteq(\mathsf{vars}_{\mathsf{S}}(sb'), l)$

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{[]}, \mathsf{refl}(\_), h_1, h_2) \equiv h_2$

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{\mathsf{var\_var}}(x, h_3), h, h_1, h_2) \equiv \mathsf{vars}_{\mathsf{lemma}}(h_3, h, \subseteq_{\mathsf{trans}}(\subseteq\mathsf{vars}_{\mathsf{var\_var}}(x, lp_1), h_1), h_2)$

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{\mathsf{var\_term}}(lp_1, h_3, h_4), h, h_1, h_2) \equiv$
case $h \in =(\mathsf{unify}(sb, \mathsf{uniacc}_{\mathsf{var\_term}}(lp_1, h_3, h_4)), \vee_{\mathsf{L}}(sb'))$ **of**
end

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{:=\mathsf{var\_term}}(h_3, h_4), h, h_1, h_2) \equiv$
$\mathsf{vars}_{\mathsf{lemma}}(h_4,$
$h,$
$\subseteq\mathsf{vars}_{:=\mathsf{LPT}}(x, lp_1, \subseteq{+}{+}_{\mathsf{redL}}(\subseteq{+}{+}_{\mathsf{redL}}(h_1)), \subseteq{+}{+}_{\mathsf{redR}}(+{+}(\mathsf{vars}_{\mathsf{T}}(\mathsf{var}(x)), \mathsf{vars}_{\mathsf{T}}(t)), h_1)),$
$\subseteq_{:}(\subseteq{+}{+}_{\mathsf{L}}(\subseteq\mathsf{vars}_{:=\mathsf{S}}(x, sb, \subseteq{+}{+}_{\mathsf{redL}}(\subseteq{+}{+}_{\mathsf{redL}}(h_1)), h_2), \subseteq{+}{+}_{\mathsf{redL}}(\subseteq{+}{+}_{\mathsf{redL}}(h_1))),$
$\subseteq\in_{\mathsf{trans}}(\subseteq{+}{+}_{\mathsf{redR}}(\mathsf{vars}_{\mathsf{T}}(t), \subseteq{+}{+}_{\mathsf{redL}}(h_1)), \in_{\mathsf{hd}}(x, [])))))$

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{\mathsf{var\_fun}}(h_3), h, h_1, h_2) \equiv$
$\mathsf{vars}_{\mathsf{lemma}}(h_3, h, \subseteq_{\mathsf{trans}}(\mathsf{fst}(\subseteq\mathsf{vars}_{\mathsf{var\_fun}}(f, x, lt, lp_1)), h_1), h_2)$

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{\mathsf{fun\_fun}}(lt_1, lt_2, lp_1, h_3), h, h_1, h_2) \equiv$
case $h \in =(\mathsf{unify}(sb, \mathsf{uniacc}_{\mathsf{fun\_fun}}(lt_1, lt_2, lp_1, h_3)), \vee_{\mathsf{L}}(sb'))$ **of**
end

$\mathsf{vars}_{\mathsf{lemma}}(\mathsf{uniacc}_{\mathsf{zip\_fun\_fun}}(f, h_3), h, h_1, h_2) \equiv$
$\mathsf{vars}_{\mathsf{lemma}}(h_3, h, \subseteq_{\mathsf{trans}}(\mathsf{fst}(\subseteq\mathsf{vars}_{\mathsf{fun\_fun}}(f, f, lt_1, lt_2, lp_1)), h_1), h_2)$

When the list of pairs of terms is empty (first equation in the proof), the resulting substitution is the accumulated substitution sb and the proof of the lemma is trivial.

The second equation considers the case where the input list of pairs of terms has the form $(x, x) : lp'$. The result of unifying the list $(x, x) : lp'$ is, by hypothesis, the substitution sb' and, by definition of the algorithm unify, equal to the result of unifying the list lp'. Hence, we have that the result of unifying the list lp' is equal to the substitution sb'. By recursion on the proof that the list lp' satisfies the predicate UniAcc (that is, the parameter h_3), we obtain a proof that the variables in the resulting substitution sb' are included in the list l. To the recursive call, we have to provide a proof that the variables in lp' are included in l, which is obtained from the fact that the variables in the list $(x, x) : lp'$ are included in l. We also have to supply a proof that the variables in the accumulated substitution are included in l, which is given by the parameter h_2.

The next equation handles the case where the input list is of the form $(x, t) : lp'$, with $x \in \mathsf{vars}_{\mathsf{T}}(t)$ and $x \neq t$. Here, h is a proof that the algorithm unify results in the substitution sb' which contradicts the fact that, by definition of the algorithm unify, this case results in the value error.

The fourth, fifth and last equation are similar to the second one. The sixth equation is similar to the third one. Notice that, as the accumulated substitution changes in the fourth equation, we have to construct a proof that the variables in the new accumulated substitution are included in the list l from the fact that both the variables in the original accumulated substitution and in the input list are included in l.

6.3 Idempotence Property

To prove that if the algorithm Unify results in a substitution, then the substitution is idempotent, we first prove a lemma stating that the resulting substitution satisfies the predicate Idem, and then we use the fact that every substitution that satisfies this predicate is idempotent.

$$\text{idempotent}_{\text{prop}} \in (=(\text{Unify}(lp), \vee_{\text{L}}(sb))) \text{ Idempotent}(sb)$$
$$\text{idempotent}_{\text{prop}}(h) \equiv \text{idem}_{\text{to}}\text{idempotent}(\text{idem}_{\text{prop}}(h))$$

To prove that if the algorithm Unify results in a substitution then the substitution satisfies the predicate Idem, we use the same technique as before. That is, we define an auxiliary lemma that takes, among other parameters, a proof that the input list satisfies the predicate UniAcc and returns a proof that the substitution that results from the algorithm unify satisfies the predicate Idem. This lemma also takes two other extra parameters: a proof that the set of variables that occur in the input list of pairs of terms and the domain of the accumulated substitution are disjoint, and a proof that the accumulated substitution satisfies the predicate Idem. When we use this lemma from the property $\text{idem}_{\text{prop}}$ the initial accumulated substitution is empty, and then it is easy to construct the proofs of these two extra parameters.

$$\text{idem}_{\text{prop}} \in (=(\text{Unify}(lp), \vee_{\text{L}}(sb))) \text{ Idem}(sb)$$
$$\text{idem}_{\text{prop}}(h) \equiv \text{idem}_{\text{lemma}}(\text{allUniAcc}_{\text{LPT}}(lp), h, \text{disj}_{[]\text{R}}(\text{vars}_{\text{LPT}}(lp)), []_{\text{idem}})$$

The proof of the auxiliary lemma uses several general lemmas about list inclusion and disjoint lists whose proofs can be found in section C.3.2. It also uses a few particular lemmas about substitutions, idempotence and lists of pairs of terms. See sections C.4.6 and C.4.4 for the proofs of these particular lemmas.

We show the proof of the auxiliary lemma in figure 6.1. This lemma is proven by recursion on the proof that the initial list satisfies the predicate UniAcc.

6.4 Most General Unifier Property

Before proving that if the unification algorithm results in a substitution, then this substitution is a most general unifier of the input list, we prove two auxiliary lemmas over the algorithm unify. Both lemmas take, among other parameters, a proof that the input list satisfies the predicate UniAcc. In the proofs of both auxiliary lemmas we use a few lemmas about unification of lists of pairs of terms and about unification of substitutions, the proofs of which can be found in sections C.4.4 and C.4.6, respectively. We show the proofs of both lemmas in figure 6.2.

The first auxiliary lemma establishes that if the input list of pairs of terms satisfies the predicate UniAcc, if a substitution sb' unifies both the input list and the accumulated substitution, and if the algorithm unify results in a substitution sb_1, then the substitution sb' also unifies the resulting substitution sb_1. The proof is made by recursion on the proof that the input list satisfies UniAcc.

The second auxiliary lemma establishes that if the input list satisfies the

$$
\begin{aligned}
&\mathrm{idem}_{\mathrm{lemma}} \in (p \in \mathrm{UniAcc}(lp);\\
&\qquad =(\mathrm{unify}(sb, p), \vee_{\mathrm{L}}(sb'));\\
&\qquad \mathrm{Disjoint}(\mathrm{vars}_{\mathrm{LPT}}(lp), \mathrm{dom}(sb));\\
&\qquad \mathrm{Idem}(sb)\\
&\qquad) \mathrm{Idem}(sb')\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{[]}, \mathrm{refl}(\_), h_1, h_2) \equiv h_2\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_var}}(x, h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(h_3, h, \subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq\mathrm{vars}_{\mathrm{var\_var}}(x, lp_1), h_1), h_2)\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_term}}(lp_1, h_3, h_4), h, h_1, h_2) \equiv\\
&\qquad \textbf{case } h \in =(\mathrm{unify}(sb, \mathrm{uniacc}_{\mathrm{var\_term}}(lp_1, h_3, h_4)), \vee_{\mathrm{L}}(sb')) \textbf{ of}\\
&\qquad \textbf{end}\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{:=\mathrm{var\_term}}(h_3, h_4), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(\\
&\qquad\quad h_4,\\
&\qquad\quad h,\\
&\qquad\quad \mathrm{disj}_{:\mathrm{R}}(=_{\mathrm{subst1}}(=_{\mathrm{dom}}(x, t, sb), \subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq\mathrm{vars}_{\mathrm{var\_term}}(x, t, lp_1), h_1)), \notin :=_{\mathrm{LPT}=\mathrm{var}}(lp_1, h_3)),\\
&\qquad\quad :_{\mathrm{idem}}(\mathrm{idem}:=(h_2, h_3, \subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq^{+}{}_{\mathrm{RR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1)),\\
&\qquad\qquad \mathrm{disj}_{:\mathrm{R}}(\\
&\qquad\qquad\quad =_{\mathrm{subst1}}(=_{\mathrm{dom}}(x, t, sb), \subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq^{+}{}_{\mathrm{RR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1)),\\
&\qquad\qquad\quad h_3),\\
&\qquad\qquad \notin :=_{\mathrm{S}}=_{\mathrm{var}}(\\
&\qquad\qquad\quad sb,\\
&\qquad\qquad\quad h_3,\\
&\qquad\qquad\quad \mathrm{disj}_{\mathrm{to}}\notin(\mathrm{disj}_{\mathrm{symm}}(\subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq^{+}{}_{\mathrm{LR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1))))))\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_fun}}(h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(h_3, h, \subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\mathrm{fst}(\subseteq\mathrm{vars}_{\mathrm{var\_fun}}(f, x, lt, lp_1)), h_1), h_2)\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, h_3), h, h_1, h_2) \equiv\\
&\qquad \textbf{case } h \in =(\mathrm{unify}(sb, \mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, h_3)), \vee_{\mathrm{L}}(sb')) \textbf{ of}\\
&\qquad \textbf{end}\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{zip\_fun\_fun}}(f, h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(h_3, h, \subseteq_{\wedge}\mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\mathrm{fst}(\subseteq\mathrm{vars}_{\mathrm{fun\_fun}}(f, f, lt_1, lt_2, lp_1)), h_1), h_2)
\end{aligned}
$$

Figure 6.1: Proof of the Auxiliary Lemma for the Idempotence Property

predicate UniAcc, if the algorithm unify results in a substitution sb_1 and if a substitution sb' unifies sb_1, then sb' also unifies both the input list of pairs of terms and the accumulated substitution. The proof is made by recursion on the proof that the input list satisfies UniAcc.

We can use these two auxiliary lemmas to prove that if the unification algorithm results in a substitution, then this substitution and the input list of pairs of terms are equivalent, in the sense that both have the same set of unifiers (as defined in section 5.3):

$$
\begin{aligned}
&\equiv_{\mathrm{Lp\_Unify}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb))) \equiv_{\mathrm{LpSb}}(lp, sb)\\
&\quad \equiv_{\mathrm{Lp\_Unify}}(h) \equiv\\
&\qquad \forall_{\mathrm{I}}([sb_1]\\
&\qquad\qquad \wedge_{\mathrm{I}}(\Rightarrow_{\mathrm{I}}([h_1]\mathrm{unifies}_{\mathrm{LpSb}}\mathrm{to}\mathrm{unifies}_{\mathrm{Sb}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), \mathrm{fst}(h_1), \mathrm{snd}(h_1)), h)),\\
&\qquad\qquad\quad \Rightarrow_{\mathrm{I}}([h_1]\mathrm{unifies}_{\mathrm{Sb}}\mathrm{to}\mathrm{unifies}_{\mathrm{LpSb}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), h_1, h))))
\end{aligned}
$$

The parameter h_1 in the first argument of the conjunction is a proof that the

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}} \in (p \in \text{UniAcc}(lp);$
$\text{unifies}_{\text{LPT}}(sb', lp);$
$\text{unifies}_{\text{S}}(sb', sb);$
$=(\text{unify}(sb, p), \vee_{\text{L}}(sb_1))$
$)\,\text{unifies}_{\text{S}}(sb', sb_1)$

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{[]}, h, h_1, \text{refl}(\_)) \equiv h_1$

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{var\_var}}(x, h_3), \text{unifies}_{\text{LPT\_:}}(h_4, h_5), h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(h_3, h_4, h_1, h_2)$

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{var\_term}}(lp_1, h_3, h_4), h, h_1, h_2) \equiv$
case $h_2 \in =(\text{unify}(sb, \text{uniacc}_{\text{var\_term}}(lp_1, h_3, h_4)), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{:=\text{var\_term}}(h_3, h_4), \text{unifies}_{\text{LPT\_:}}(h_5, h_6), h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(h_4, \text{unifies}_{\text{LPT\_:=R}}(\text{unifies}_{\text{LPT\_:}}(h_5, h_6)), \text{unifies}_{\text{S\_:\_:=R}}(h_6, h_1), h_2)$

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{var\_fun}}(h_3), h, h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(h_3, \text{unifies}_{\text{LPT\_fvtovf}}(h), h_1, h_2)$

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{L}}(\Rightarrow_{\text{I}}(f_1))), h, h_1, h_2) \equiv$
case $h_2 \in =(\text{unify}(sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{L}}(\Rightarrow_{\text{I}}(f_1)))), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{R}}(\Rightarrow_{\text{I}}(f_1))), h, h_1, h_2) \equiv$
case $h_2 \in =(\text{unify}(sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{R}}(\Rightarrow_{\text{I}}(f_1)))), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(\text{uniacc}_{\text{zip\_fun\_fun}}(f, h_3), h, h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbto}}\text{unifies}_{\text{Sb}}(h_3, \text{unifies}_{\text{LPT\_funtozip}}(h), h_1, h_2)$

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}} \in (p \in \text{UniAcc}(lp);$
$\text{unifies}_{\text{S}}(sb', sb_1);$
$=(\text{unify}(sb, p), \vee_{\text{L}}(sb_1))$
$)\wedge(\text{unifies}_{\text{LPT}}(sb', lp), \text{unifies}_{\text{S}}(sb', sb))$

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{[]}, h, \text{refl}(\_)) \equiv \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_[]}}(sb'), h)$

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{\text{var\_var}}(x, h_2), h, h_1) \equiv$
case $\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(h_2, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', lp_1), \text{unifies}_{\text{S}}(sb', sb))$ **of**
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_:}}(h_3, \text{refl}(\text{appP}_{\text{T}}(sb', \text{var}(x)))), h_4)$
end

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{\text{var\_term}}(lp_1, h_2, h_3), h, h_1) \equiv$
case $h_1 \in =(\text{unify}(sb, \text{uniacc}_{\text{var\_term}}(lp_1, h_2, h_3)), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{:=\text{var\_term}}(h_2, h_3), h, h_1) \equiv$
case $\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(h_3, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', :=_{\text{LPT}}(x, t, lp_1)),$ **of**
$\text{unifies}_{\text{S}}(sb', :(:=_{\text{S}}(x, t, sb), .(x, t))))$
$\wedge_{\text{I}}(h_4, \text{unifies}_{\text{S\_:}}(h_6, h_7)) \Rightarrow$
$\wedge_{\text{I}}(\text{unifies}_{\text{LPT\_:}}(\text{unifies}_{\text{LPT\_:=L}}(lp_1, h_7, h_4), h_7), \text{unifies}_{\text{S\_:=L}}(sb, h_7, h_6))$
end

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{\text{var\_fun}}(h_2), h, h_1) \equiv$
case $\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(h_2, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', :(lp_1, .(\text{var}(x), \text{fun}(f, lt)))),$ **of**
$\text{unifies}_{\text{S}}(sb', sb))$
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_vftofv}}(h_3), h_4)$
end

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, h_2), h, h_1) \equiv$
case $h_1 \in =(\text{unify}(sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, h_2)), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(\text{uniacc}_{\text{zip\_fun\_fun}}(f, h_2), h, h_1) \equiv$
case $\text{unifies}_{\text{Sbto}}\text{unifies}_{\text{LpSb}}(h_2, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', ++(\text{zip}(lt_1, lt_2), lp_1)),$ **of**
$\text{unifies}_{\text{S}}(sb', sb))$
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_ziptofun}}(f, h_3), h_4)$
end

Figure 6.2: Auxiliary Lemmas for the Most General Unifier Property

substitution sb_1 unifies both the list lp and the empty substitution, and the parameter h_1 in the second argument of the conjunction is a proof that the substitution sb_1 unifies the substitution sb.

To prove the most general unifier property, we make use of the following two lemmas. The first lemma states that if a substitution satisfies the predicate Idem, then the substitution unifies itself. The second lemma says that if a substitution sb' satisfies the predicate Idem and if there is a substitution sb that unifies sb', then the substitution sb' is at least as general as the substitution sb (as defined in section 5.3).

$$\mathrm{idem}_{\mathrm{to}}\mathrm{unifies} \in (\mathrm{Idem}(sb))\ \mathrm{unifies}_{\mathrm{S}}(sb, sb)$$
$$\mathrm{unifies}_{\wedge}\mathrm{idem}_{\mathrm{to}}\mathrm{mgu}_{\mathrm{aux}} \in (\mathrm{Idem}(sb');\ \mathrm{unifies}_{\mathrm{S}}(sb, sb')) \leq_{\mathrm{Sb}}(sb', sb)$$

Both lemmas are proven by recursion on the proofs that the substitutions sb and sb', respectively, satisfy the predicate Idem. See section C.4.6 for the complete proofs of these lemmas.

These two auxiliary lemmas are used for proving the following two properties, which are needed to prove the most general unifier property.

The first property states that if the unification algorithm results in a substitution, then this substitution unifies the input list of pairs of terms.

$$\mathrm{unifies}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb)))\ \mathrm{unifies}_{\mathrm{LPT}}(sb, lp)$$
$$\mathrm{unifies}_{\mathrm{prop}}(h) \equiv \mathrm{fst}(\mathrm{unifies}_{\mathrm{Sb}\mathrm{to}}\mathrm{unifies}_{\mathrm{LpSb}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), \mathrm{idem}_{\mathrm{to}}\mathrm{unifies}(\mathrm{idem}_{\mathrm{prop}}(h)), h))$$

The second property states that if the unification algorithm results in a substitution sb and if a substitution sb' unifies the input list, then sb is more general than sb'.

$$\mathrm{mgu}_{\mathrm{prop\_aux}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb));\ \mathrm{unifies}_{\mathrm{LPT}}(sb', lp)) \leq_{\mathrm{Sb}}(sb, sb')$$
$$\mathrm{mgu}_{\mathrm{prop\_aux}}(h, h_1) \equiv$$
$$\mathrm{unifies}_{\wedge}\mathrm{idem}_{\mathrm{to}}\mathrm{mgu}_{\mathrm{aux}}(\mathrm{idem}_{\mathrm{prop}}(h),$$
$$\mathrm{unifies}_{\mathrm{LpSb}\mathrm{to}}\mathrm{unifies}_{\mathrm{Sb}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), h_1, \mathrm{unifies}_{\mathrm{S\_[]}}(sb'), h))$$

Finally, we prove the most general unifier property, in other words, we prove that if the unification algorithm results in a substitution, then this substitution is a most general unifier of the input list of pairs of terms. For proving this proposition, we use the properties $\mathrm{unifies}_{\mathrm{prop}}$ and $\mathrm{mgu}_{\mathrm{prop\_aux}}$.

$$\mathrm{mgu}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb)))\ \mathrm{mgu}(sb, lp)$$
$$\mathrm{mgu}_{\mathrm{prop}}(h) \equiv \wedge_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{prop}}(h), \forall_{\mathrm{I}}([sb'] \Rightarrow_{\mathrm{I}}([h_1]\mathrm{mgu}_{\mathrm{prop\_aux}}(h, h_1))))$$

Chapter 7

The Integrated Approach

In the last two chapters, we have introduced the formalisation of the unification algorithm in type theory and we have presented several proofs that show the partial correctness of the algorithm. That is, we showed that the unification algorithm returns the value error only if there exists no substitution that unifies the input list of pairs of terms; otherwise it returns an idempotent substitution that is a most general unifier of the input list of pairs of terms and whose variables are included in the list of variables of the input list of pairs of terms. We have presented the algorithm and each of the proofs in an separate way, that is, we first presented the algorithm and then we proved the desired properties one by one. This methodology is known as the *external approach*.

However, we can "integrate" all the desired properties into the complete specification for the unification algorithm, and then given a proof that there is an object that satisfies this specification. Such an object will have a unification algorithm embedded. This methodology is known as the *integrated approach* or the *internal approach*.

In this chapter, we present the type theory formalisation of the integrated approach to the unification algorithm which has the following specification:

> Given a list of pairs of terms lp, either there does not exist any substitution that unifies the list lp, or there exists a substitution sb such that the variables that occur in sb are included in the variables that occur in the list lp, sb is idempotent and it is a most general unifier of the input list lp.

In order to formalise the specification, we first introduce the definition of a constructor that defines the conjunction of three propositions:

$$\wedge_3 \in (A, B, C \in \mathbf{Set})\ \mathbf{Set}$$
$$\wedge_{I3} \in (a \in A; b \in B; c \in C)\ \wedge_3(A, B, C)$$

Using this constructor, the type of the main theorem we prove here is the following:

$$\begin{aligned}\text{Theorem} \in\ & (lp \in \text{ListPT}\\ &)\vee(\neg(\exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp))),\\ & \quad \exists(\text{Subst}, [sb]\wedge_3(\subseteq(\text{vars}_{\text{S}}(sb), \text{vars}_{\text{LPT}}(lp)), \text{Idempotent}(sb), \text{mgu}(sb, lp))))\end{aligned}$$

To prove this theorem we use an auxiliary theorem that takes a proof that the input list satisfies the predicate UniAcc as a parameter. This auxiliary theorem has the following type:

$$\begin{aligned}\text{Th} \in\ & (\text{UniAcc}(lp)\\ &)\vee(\neg(\exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp))),\\ & \quad \exists(\text{Subst}, [sb]\wedge_3(\subseteq(\text{vars}_{\text{S}}(sb), \text{vars}_{\text{LPT}}(lp)), \text{Idem}(sb), \text{mgu}(sb, lp))))\end{aligned}$$

Notice that the result of this auxiliary theorem is very similar to the result of the main theorem. The only difference is that in the auxiliary theorem we ask the substitution (when it exists) to satisfy the predicate Idem while in the main theorem we ask the substitution to be idempotent.

The proof of the main theorem is immediate from the auxiliary theorem. When we obtain a proof that there exists a substitution that unifies the input list of pairs of terms, we have to take the proof that this substitution satisfies the predicate Idem into a proof that the substitution is idempotent.

$$\begin{aligned}&\text{Theorem} \in (lp \in \text{ListPT}\\ &\qquad)\vee(\neg(\exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp))),\\ &\qquad\quad \exists(\text{Subst}, [sb]\wedge_3(\subseteq(\text{vars}_{\text{S}}(sb), \text{vars}_{\text{LPT}}(lp)), \text{Idempotent}(sb), \text{mgu}(sb, lp))))\\ &\quad \text{Theorem}(lp) \equiv\\ &\qquad \textbf{case}\ \text{Th}(\text{allUniAcc}_{\text{LPT}}(lp)) \in \vee(\neg(\exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp))),\\ &\qquad\qquad\qquad\qquad \exists(\text{Subst},\\ &\qquad\qquad\qquad\qquad\quad [sb]\wedge_3(\subseteq(\text{vars}_{\text{S}}(sb), \text{vars}_{\text{LPT}}(lp)), \text{Idem}(sb), \text{mgu}(sb, lp))))\\ &\qquad\quad \vee_{\text{L}}(h) \Rightarrow \vee_{\text{L}}(h)\\ &\qquad\quad \vee_{\text{R}}(\exists_{\text{I}}(sb, \wedge_{\text{I3}}(h_1, h_2, h_3))) \Rightarrow \vee_{\text{R}}(\exists_{\text{I}}(sb, \wedge_{\text{I3}}(h_1, \text{idem}_{\text{to}}\text{idempotent}(h_2), h_3)))\\ &\qquad \textbf{end}\end{aligned}$$

The auxiliary theorem is proven by recursion on the proof that the input list of pairs of terms satisfies the predicate UniAcc. As we prove all the desired properties in a single theorem, the proof of this theorem is two pages long. See section C.7 for the complete ALF proof of this auxiliary theorem.

In the equations that correspond to the cases where the algorithm unify is defined by recursion, we consider cases on the result of the recursive calls.

For the proofs that there exists no substitution that unifies the input list, we use a few lemmas whose ALF proofs can be found in section C.7. There are two kinds of these lemmas: those that take a proof that there does not exist a substitution that unifies the list on which we perform the recursion into a proof that there does not exist a substitution that unifies the input list (second, fourth, fifth and last equations in the proof of the auxiliary theorem), and those that take the proofs that state certain conditions on the input list into a proof that there does not exist a substitution that unifies such a list of pairs of terms (third and sixth equations in the proof of the auxiliary theorem).

In six of the seven equations of the proof of this theorem (one equation for each of the seven constructors of the predicate UniAcc), no surprises arise. These are the cases where the accumulated substitution does not change in the definition of the algorithm unify and the cases where the algorithm unify gives a

basic result (either a substitution that unifies the input list of pairs of terms or a proof that such a substitution does not exist). Moreover, most of the lemmas we use to prove these cases were already used in the different proofs we gave in the previous chapter.

However, it is interesting to study the case where the input list has the form $(x, t) : lp$ with $x \notin_{\mathrm{L}} \mathrm{vars}_{\mathrm{T}}(t)$, which is the only case in the definition of the algorithm **unify** where the accumulated substitution changes. Below, we show the part of the proof that corresponds to this case:

$$
\begin{array}{l}
\mathrm{Th}(\mathrm{uniacc}_{:=\mathrm{var\_term}}(h_1, h_2)) \equiv \\
\quad \textbf{case}\ \mathrm{Th}(h_2) \in \vee(\neg(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, :=_{\mathrm{LPT}}(x, t, lp_1))))),\ \textbf{of} \\
\qquad\qquad \exists(\mathrm{Subst}, \\
\qquad\qquad\quad [sb]\wedge_3(\subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}(:=_{\mathrm{LPT}}(x, t, lp_1))), \\
\qquad\qquad\qquad\quad \mathrm{Idem}(sb), \\
\qquad\qquad\qquad\quad \mathrm{mgu}(sb, :=_{\mathrm{LPT}}(x, t, lp_1))))) \\
\qquad \vee_{\mathrm{L}}(h) \Rightarrow \vee_{\mathrm{L}}(\Rightarrow_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_var\_term}:=_{\mathrm{to}}\perp}(h_1, h))) \\
\qquad \vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb, \wedge_{\mathrm{I3}}(h, h_3, \wedge_{\mathrm{I}}(h_4, h_5)))) \Rightarrow \\
\qquad\quad \vee_{\mathrm{R}}(\exists_{\mathrm{I}}(:(sb, .(x, \mathrm{appP}_{\mathrm{T}}(sb, t))), \\
\qquad\qquad \wedge_{\mathrm{I3}}(\subseteq{+}{+}_{\mathrm{L}}(\subseteq_{\mathrm{trans}}(h, \subseteq\mathrm{vars}_{\mathrm{var\_term}}(x, t, lp_1)), \\
\qquad\qquad\qquad \subseteq{:}(\subseteq_{\mathrm{trans}}(\subseteq\mathrm{vars}_{\mathrm{appP\_T}}(sb, t), \\
\qquad\qquad\qquad\qquad \subseteq{+}{+}_{\mathrm{L}}(\subseteq_{\mathrm{trans}}(h, \subseteq\mathrm{vars}_{\mathrm{var\_term}}(x, t, lp_1)), \\
\qquad\qquad\qquad\qquad\quad \subseteq{+}{+}_{\mathrm{RR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)))), \\
\qquad\qquad\qquad\quad \in{+}{+}_{\mathrm{monL}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \in{+}{+}_{\mathrm{monR}}(\mathrm{vars}_{\mathrm{T}}(t), \in_{\mathrm{hd}}(x, [])))))), \\
\qquad\qquad \mathrm{idem}_{:}(h_1, \subseteq_{\wedge\notin_{\mathrm{to}}\notin}(h, \notin_{:=\mathrm{LPT}=\mathrm{var}}(lp_1, h_1)), h_3), \\
\qquad\qquad \wedge_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_:}}(\\
\qquad\qquad\quad \mathrm{unifies}_{\mathrm{LPT}:=_{\mathrm{to}}}\mathrm{unifies}_{\mathrm{LPT}}(h_1, \subseteq_{\wedge\notin_{\mathrm{to}}\notin}(h, \notin_{:=\mathrm{LPT}=\mathrm{var}}(lp_1, h_1)), h_3, h_4), \\
\qquad\qquad\quad =_{\mathrm{var\_term}}\mathrm{appP}_{\mathrm{T}}(h_1, \subseteq_{\wedge\notin_{\mathrm{to}}\notin}(h, \notin_{:=\mathrm{LPT}=\mathrm{var}}(lp_1, h_1)), h_3)), \\
\qquad\qquad\quad \forall_{\mathrm{I}}([sb'] \\
\qquad\qquad\qquad \Rightarrow_{\mathrm{I}}([h_6]\exists_{\mathrm{I}}(\mathrm{witness}(\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(h_5, sb'), \mathrm{unifies}_{\mathrm{LPT\_:=R}}(h_6))), \\
\qquad\qquad\qquad\quad \forall_{\mathrm{I}}([t']\mathrm{mgu}_{\mathrm{auxT}}(\\
\qquad\qquad\qquad\qquad t', \\
\qquad\qquad\qquad\qquad \mathrm{idem}_{:}(h_1, \\
\qquad\qquad\qquad\qquad\quad \subseteq_{\wedge\notin_{\mathrm{to}}\notin}(h, \notin_{:=\mathrm{LPT}=\mathrm{var}}(lp_1, h_1)), \\
\qquad\qquad\qquad\qquad\quad h_3), \\
\qquad\qquad\qquad\qquad h_6, \\
\qquad\qquad\qquad\qquad \mathrm{proof}(\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(h_5, sb'), \mathrm{unifies}_{\mathrm{LPT\_:=R}}(h_6)))))))))))) \\
\quad \textbf{end}
\end{array}
$$

The parameter h_2 is a proof that the list $lp'\,[x{:=}t]$ satisfies the predicate UniAcc. By recursion on this proof, we obtain that either there exists no substitution that unifies the list $lp'\,[x{:=}t]$, or there exists a substitution sb such that $\mathrm{vars}_{\mathrm{S}}(sb) \subseteq \mathrm{vars}_{\mathrm{LPT}}(lp'\,[x{:=}t])$, it satisfies the predicate Idem and it is a most general unifier of the list $lp'\,[x{:=}t]$.

From the proof that there exists no substitution that unifies the list $lp'\,[x{:=}t]$, it is easy to obtain a proof that there exists no substitution that unifies the input list $(x, t) : lp'$.

Otherwise, given sb that unifies $lp'\,[x{:=}t]$, we have to give a substitution that unifies $(x, t) : lp'$ and that also satisfies the desired properties. As we have that $\mathrm{vars}_{\mathrm{S}}(sb) \subseteq \mathrm{vars}_{\mathrm{LPT}}(lp'\,[x{:=}t])$ and since $x \notin_{\mathrm{L}} \mathrm{vars}_{\mathrm{LPT}}(lp'\,[x{:=}t])$ because $x \notin_{\mathrm{L}} \mathrm{vars}_{\mathrm{T}}(t)$, we know that $x \notin_{\mathrm{L}} \mathrm{vars}_{\mathrm{S}}(sb)$. Hence, there is no need to substitute t

for x in sb since $(sb\,[x{:=}t]) = sb$. Thus, the unifier we give here is the substitution $(x,\, sb(t)) : sb$ and we have to supply proofs that this substitution satisfies the required properties. As the way we construct this unifier is different from the way we construct the unifier in the algorithm unify, we need a few new lemmas that are just used in the part of the proof showed above. The proofs of those lemmas can be found in sections C.4.4, C.4.6 and C.7.

Finally, it is interesting to notice that even though the unifier that results from the algorithm unify and the unifier that results from the theorem we prove in this chapter are constructed in different ways, actually both produce the same substitution. The difference in the construction arises from the fact that both unifiers are built in reverse sequences. In the algorithm unify, given an input list of the form $(x,\, t) : lp$ with $x \notin_{\mathrm{L}} \mathsf{vars}_{\mathrm{T}}(t)$ and an accumulated substitution sb, we add the pair $(x,\, t)$ to the substitution $sb\,[x{:=}t]$ and we continue looking for a unifier for the list $lp\,[x{:=}t]$. However, if sb is the resulting unifier for the list $lp\,[x{:=}t]$ in the theorem we prove in this chapter, we add the pair $(x,\, sb(t))$ to it. In this way, both unifiers are constructed in reverse order, but the terms associated to each of the variables in the domain of the substitution are the same. However, the fact that both unifiers are constructed in a different way is not a consequence of whether we decide to use the external or internal approach to formalise the unification problem but of the choices we made when defining both the predicate UniAcc and the specification of the internal approach.

Chapter 8

Conclusions

Here, we present some conclusions, related work and future work.

The complete ALF formalisation of this work, that we present in appendix C, consists of 30 files with more than 330 functions that require approximately 180 Kbytes. After working out the details on paper, it took about three weeks to formalise the functions in ALF. Approximately half of this time was used for defining the UniAcc predicate and the proof that all the lists of pairs of terms satisfy this predicate, and the other half for defining the proof that shows the partial correctness of the unification algorithm. Unfortunately, as there are no good general libraries for ALF, we had to start our work from scratch. For this reason, most of the time spent in the proof that all the lists of pairs of terms satisfy the predicate UniAcc was used for defining lemmas about inequality of natural number, inclusion of lists and other general functions.

Our work is heavily based on inductive families, for which ALF is very suitable. Unfortunately, ALF is not maintained anymore, which makes its use a bit risky since no support is available if something goes wrong. Its successor Agda [Coq98] does not allow the definition of predicates like our special-purpose accessibility predicate UniAcc, and then it would not have been possible to use Agda to formalise the unification algorithm following our approach. On the other hand, we believe that this formalisation of the unification algorithm can also be performed in other proof assistants that allow the definition of inductive predicates like our special-purpose accessibility predicate, as for example the proof assistant Coq [DFH$^+$91].

The code of the algorithm we obtain by using the UniAcc predicate to formalise the unification algorithm is short and elegant. The UniAcc predicate contains all the information needed in order to handle the recursive calls, and it is exactly this fact what makes the UniAcc predicate appropriate for the formalisation of the unification algorithm in type theory. On the other hand, the standard accessibility predicate, even if useful in other cases, turned out not to be a good solution for our particular case study. The unification algorithm that results from using the standard accessibility predicate to handle the recursive calls is much longer and more complicated than the one obtained by using the

predicate UniAcc. Besides, the formalisation that uses the standard accessibility predicate contains big parts of code that are computationally irrelevant which we, as programmers, are not interested in having together with the code of the algorithm. These parts include the proofs that the new lists to be unified in the recursive calls are smaller than the original list of pairs of terms.

Together with the UniAcc predicate and our type theory version of the unification algorithm, we present a proof that shows that our special-purpose accessibility predicate holds for any possible input list of pairs of terms. This proof, which is presented in appendix A, has a similar skeleton to the type theory version of the unification algorithm that uses the standard accessibility predicate to handle the recursive calls. Observe that this proof is actually the only place in our solution where we need to mention the proofs that the new lists to be unified in the recursive calls are smaller than the original list of pairs of terms.

The proof that shows the partial correctness of the unification algorithm considers the same cases as the ones studied when formalising the algorithm. Hence, the different proofs that we presented in sections 6 and 7 have also benefitted from the way we defined the predicate UniAcc and the unification algorithm. Most of these proofs are defined by recursion on the proof that the input list satisfies the predicate UniAcc, and they are short and concise. On the other hand, if we would have defined the unification algorithm by using the standard accessibility predicate, each of the proofs would have had the same big skeleton as the algorithm, which implies that they would have been much longer than the ones we presented in this work. Besides, in each of these proofs, we would have had to mention the proofs that the new lists to be unified in the recursive calls are smaller than the original list of pairs of terms.

In addition, observe that there is actually not so much difference between the proofs we present in chapters 6 and 7 (see sections C.6 and C.7 respectively, for the ALF codes of the formalisations of the proofs). As each of the different proofs is constructed by recursion on our special-purpose accessibility predicate, each proof considers seven cases, one for each of the seven introduction rules of the predicate. In most of the cases, the proof of the theorem presented in chapter 7 practically consists of gathering together the different lemmas used for proving the different properties presented in chapter 6. The only case where the proofs presented in chapters 6 and 7 differ is in the case where the resulting substitution actually changes. As already mentioned, even though the resulting unifiers are the same in both the external and internal approach, they are computed in reverse order. As we also said before, this is not a consequence of the approach used but of the choices made when defining both the predicate UniAcc and the specification of the internal approach.

Finally, we discuss a methodology that would allow us to extract Haskell programs from the type theory programs that are defined by using a special-purpose accessibility predicate to handle the recursive calls. We believe that this methodology for program extraction is easy to program, and then it can be added as part of a future program extraction module for ALF.

To summarise, we believe that the methodology we present here for formalising the unification algorithm in Martin-Löf's type theory is simple and therefore

easy to perform. Besides, it allows us to obtain short and elegant results when we use our special-purpose accessibility predicate for defining both the algorithm and the proof of its partial correctness. In addition, the same methodology can be used for writing other total and general recursive algorithms in type theory. In this way, we think that this methodology gives a step towards closing the existing gap between programming in a Haskell-like programming language and programming in Martin-Löf's type theory. However, the generalisation of this methodology to all total and general recursive algorithms remains to be studied.

8.1 Related Work

Unification has became widely known since Robinson [Rob65] used it as the central step of the inference principle called resolution. Afterwards, unification algorithms have been the centre of several studies.

In [MM82], Martelli and Montanari describe the unification problem in first-order predicate calculus as the solution of a set of equations, and give a non-deterministic unification algorithm together with the proof of its correctness. From this non-deterministic algorithm, they derive a new and efficient unification algorithm. The unification algorithm we presented in this work is a deterministic version of the first (non-deterministic) algorithm presented by Martelli and Montanari. Our mapping $\mathsf{LPT_{to}N3}$, used for proving the termination of our algorithm, is a simplification of the mapping F presented in [MM82] to show the termination of their (non-deterministic) algorithm. In addition, the notion of equivalence between lists of pairs of terms and substitutions, that we introduced in section 5.3, is an adaptation to our algorithm of the notion of equivalence of sets of equations presented in [MM82].

The deductive synthesis of the unification algorithm by Manna and Waldinger [MW81] is also a well known work on unification. Given a high-level specification of the unification algorithm, Manna and Waldinger prove a theorem that establishes the existence of an object satisfying the specification. As the proof is constructively done, the desired program can be extracted from it. Although their work is very detailed and easy to follow, it has been done completely manually and hence it has not been machine-checked. As the recursion in the program that would ultimately be extracted from their proof is not on structurally smaller elements, their work cannot be directly translated into Martin-Löf's type theory since, as we already mention it, there is no direct way of formalising general recursion in type theory.

Eriksson [Eri84] synthesises a unification algorithm from a formal specification in first-order logic with equality. Eriksson developed the proofs by hand and verified them by machine. The method guarantees partial correctness, that is, if the program finds a most general unifier of two terms, then it can be proven from the specification that the terms were unifiable. However, total correctness is not proven, which amounts to showing that if two terms are unifiable, then the program will find a most general unifier for the terms. The derived algorithm, which is expressed in a Prolog-like [CM81] style, does not report when

two terms cannot be unified and its termination has not been studied. The specification presented by Eriksson does not establish if the resulting substitution is idempotent nor if the variables that occur in it are those already occurring in the input terms.

In [Pau85], Paulson closely follows the work in [MW81] to verify the unification algorithm in LCF [GMW79]. However, although Manna and Waldinger synthesise a program, Paulson states the unification algorithm and then proves it. We can distinguish several differences between Paulson's approach and ours. While we represent terms of the form $f(t_1, \ldots, t_n)$ as the application of the function f to the list of terms $[t_1, \ldots, t_n]$, Paulson represents them as their curried version $((f(t_1))(t_2)\ldots)(t_n)$. Although this simplifies the unification algorithm a bit, it complicates the representation of terms where the function to be applied has a large arity. Paulson states that he first reformulated the work in [MW81] to use lists of variables instead of the (mathematical) notion of set, but that reasoning about lists of variables was awkward and he did not attempt to finish the formalisation using them. Thus, Paulson introduces sets as *quotient types* by defining them as equivalence classes of finite lists where the order and multiplicity of elements is ignored. Since it is not possible to do this in a simple way in Martin-Löf's type theory, we used lists of variables for our formalisation. Although it was not very straightforward to work with lists, we managed to go through without big problems. In [MW81], Manna and Waldinger forbid trivial substitution like $[(x,\ x)]$ or ambiguous ones like $[(x,\ t_1),(x,\ t_2)]$. Hence, Paulson identifies $[(x,\ x)]$ with the empty substitution $[]$ and $[(x,\ t_1),(x,\ t_2)]$ with $[(x,\ t_1)]$. As the substitutions that result from our unification algorithm are idempotent, we know that each variable appears at most once in the domain of these substitutions. In addition, given an idempotent substitution sb, we can prove that if the variable x belongs to the domain of sb, then $sb(x) \neq x$. Hence, we know that the substitutions that result from our unification algorithm are neither ambiguous nor contain trivial pairs. Paulson defines the notion of proper subterm as a function returning a value in the boolean domain. Although the normal way to define it in Martin-Löf's type theory would be as an inductively defined relation, we did not need to define this relation as we already explained in section 5.4. Since types denote domains in LCF and not sets, Paulson has to deal with numerous *definedness assertions* (as he called them) of the form $t \not\equiv \bot$ in order to avoid left sides of the equations to overlap, which would lead to contradictions. We think that the presence of these assertions everywhere in the theories makes its reading a bit heavy. Finally, the recursive calls in the unification algorithm defined by Paulson are not on structurally smaller elements. Hence, termination is not guaranteed and he proves it separately. This way of defining the algorithm is not possible in Martin-Löf's type theory, that is, defining an algorithm where the recursive calls are on non-structurally smaller elements, and it is actually the main motivation for the methodology we introduce in this work.

Rouyer has presented a verification of a first-order unification algorithm in the calculus of construction with inductive types [CP90, PM93] using the Coq proof assistant [DFH+91] (see [Rou92] for the complete French technical report

of the verification and [RL] for the English summary of the French report). Several differences distinguish our work from the one presented in [Rou92]. As the use of data types with dependent types does not allow program extraction in Coq, it is not possible for Rouyer to define the set of terms in the way we have done it in this work. Hence, Rouyer defines an extended notion called *quasi-terms* and defines terms as specific quasi-terms. Rouyer's main theorem states that any two quasi-terms either have a most general unifier that is idempotent and whose variables are included in the variables that occur in the two quasi-terms, or are not unifiable. Rouyer proves that if terms are unifiable as quasi-terms, their most general unifier is a substitution that maps terms to terms. In this way, a unification algorithm for quasi-terms yields a unification algorithm for terms. The unification algorithm presented in [Rou92] is defined by induction on the number of different variables in the input quasi-terms, and then by induction on both quasi-terms. In our opinion, one needs deep knowledge of the Coq system to understand the algorithm that underlies the main theorem presented in [Rou92]. We believe that this is due to the fact that in Coq lemmas are proven by giving a sequence of tactics instead of by directly constructing the proof object. Fortunately, the explanations given in [Rou92] and [RL] guide us in understanding the underlying algorithm by showing the different cases we need to consider in order to prove the main theorem. In Coq, associated with each inductive relation one only has the elimination rules. Proving properties from an inductive relation is not always easy, while it is usually easier to prove properties from the recursive version of the relation. Thus, for each relation, Rouyer defines both the inductive and the recursive version of the relation, and then proves that both definitions are equivalent. Due to the pattern matching facility, only the inductive definition of a relation is sufficient in ALF, which makes the whole proof a bit simpler. Another difference between our approach and Rouyer's is that to show that the terms x and t are not unifiable when $x \in \mathsf{vars}_{\mathrm{T}}(t)$, Rouyer follows the standard proof that uses the notion of proper subterms while we skip the definition of this relation as we already discussed in section 5.4. On the other hand, both Rouyer's formalisation and ours use lists of variables to formalise the set of variables in a (quasi-)term. It seems we both had to deal with similar problems due to this choice.

Jaume [Jau97] presents a formalisation of a unification algorithm for first-order terms in the calculus of construction with inductive types built from the proof of the unification algorithm for first-order quasi-terms presented in [Rou92]. The technique used by Jaume is based on defining a bijection between terms and the subset of quasi-terms that represents terms (which is defined by a predicate) and proving the preservation of the unification property. In this way, the unification algorithm is transposed from quasi-terms to terms. In other words, Jaume proves the unification property without really dealing with unification theory. As no program has been extracted from this proof, it is not possible to compare Jaume's work with ours from a programming point of view, which is actually one of our main interests.

In [RRAHM99], Ruiz-Reina et al describe a formalisation and mechanical verification of a unification algorithm using the Boyer-Moore logic [BM79] and

its theorem prover. The Boyer-Moore theorem prover is automatic in the sense that once the command to prove lemmas is invoked, the user can no longer interact with the system. On the other hand, the user can give definitions and prove lemmas to be used in later proofs, and can give "hints" to the prover when invoking the command to prove lemmas. To guarantee termination when defining recursive functions, an ordinal measure that decreases in each recursive call should be provided. Since Ruiz-Reina et al also follow the work by Martelli and Montanari, the unification algorithm formalised in [RRAHM99] is very similar to ours and most of the lemmas proven in [RRAHM99] are also proven in our work. However, the language used in their formalisation is very different from the Martin-Löf's type theory. Boyer-Moore logic is a quantifier-free first-order logic with equality that uses a language very similar to pure Lisp. While Martin-Löf's type theory is a strongly typed language, the language in Boyer-Moore logic has a very poor notion of type. In [RRAHM99], terms (the same applies for list of terms and substitution) are not defined by using any specific predicate or data type. Instead, any object in the logic can represent a term by following certain conventions. Although the objects of the logic that do not follow the conventions do not represent well-formed terms, the results in [RRAHM99] are also proven for these non well-formed terms. As the notion of type is very poor in their formalisation language, some trick is needed in order to know whether an object list in their logic represents a term or a list of terms, and then a boolean flag is used for this purpose. Many functions in the formalisation presented in [RRAHM99] return either the desired value (for example a substitution that unifies two terms), or the logical value `F` if it is not possible to obtain such a value. Then, the result of those functions can be used both for actual computations or for boolean tests, as in `if solved (second solved) F`. We believe that this untyped work methodology is not very clean and it is very easy to make small mistakes which are very difficult to find out. Finally, the transformation rules to be applied to the pairs of terms in every recursive call of the unification algorithm are defined using a selection function (that selects the pair of terms to be considered in each call) that is partially defined. Then, Ruiz-Reina et al say that the transformation rules are applied in a *non-deterministic* way. However, this is not the standard notion of non-determinism because to actually have a unification algorithm, a specific selection function that satisfies the partial definition should be provided.

Finally, McBride [McB99] has also formalised a unification algorithm in Lego [LP92]. In the formalisation, McBride exploits the use of dependent types in programming by indexing the set of terms by an upper bound on the number of different variables in the terms. In addition, the set of substitutions is also indexed by the number of different variables both in the domain and in the range of a substitution. The way in which terms and substitutions are defined permits a reformulation of the unification problem in a structural way with a lexicographic recursive structure, first over the number of different variables and then over one of the terms to be unified. In this way, McBride does not need to impose either an external termination ordering or an accessibility argument. In [McB99], McBride proves that the resulting substitution is actually a most

general unifier of the two input terms but he does not (explicitly) prove that the resulting substitution is idempotent and that it only uses variables that occur in the input terms. However, this last property can be easily inferred from the type of the resulting substitution since the types of both terms and substitutions contain information about the variables in a term and in the domain and range of a substitution. Notice that, as the domain and range of a substitution may differ, it is not possible to define the usual notion of idempotence since we cannot compose substitutions in the traditional way. However, there is another property introduced in [McB99] that does the same job as idempotence and that forms the basis of the proof of one of the correctness cases. Thus, the unifiers that result from the algorithm in [McB99] are idempotent although McBride does not explicitly prove this fact. We think the result presented in McBride's work is very interesting since it shows the importance of dependent types in programming (see [wor99] for more examples of dependent types in programming). Unfortunately, programming with dependent types is still not a normal practice, and then the program associated with McBride's formalisation is not very practicable yet, thought we hope it will be in a near future.

8.2 Future Work

There are two possible directions that we would like to pursue.

The first is related to the methodology that we used for defining the predicate UniAcc. We believe that this methodology can be used for formalising other algorithms that are total and where the recursion is on non-structurally smaller arguments, such as the Quicksort algorithm. Hence, following the same methodology used for defining the predicate UniAcc, we can define a predicate QuickAcc that contains the necessary information to handle the recursive calls for this particular sorting algorithm.

However, this methodology cannot be used when we have nested recursive calls. Consider, for example, the version of the unification algorithm for the case where the terms are either variables or binary terms of the form $t_1 \to t_2$. The Haskell version for this unification algorithm is:

```
unify_h :: Term -> Term -> Maybe Subst
unify_h (Var x) (Var y) | x == y        = []
unify_h (Var x) t | x `elem` (varsT t)  = Nothing
                  | otherwise           = Just [(x,t)]
unify_h t (Var x)                       = unify_h (Var x) t
unify_h (t1 -> t2) (t3 -> t4) =
    case unify_h t1 t3 of
         Nothing  -> Nothing
         Just sb1 -> case unify_h (appP sb1 t2) (appP sb1 t4) of
                          Nothing  -> Nothing
                          Just sb2 -> Just (app_conc sb2 sb1)
```

where the function `app_conc` concatenates the first substitution with the result

of applying the first substitution to all the terms in the second one.

Now, when we know that the result of unifying the terms t_1 and t_2 is not an error, the call

```
unify_h (appP sb1 t2) (appP sb1 t4)
```

is actually the same as the call

```
unify_h (appP (unJust (unify_h t1 t3)) t2)
        (appP (unJust (unify_h t1 t3)) t4)
```

where `unJust (Just sb) = sb`. If we follow the methodology described in section 4.5.1 we obtain a special-purpose predicate where the result of the unification algorithm appears in the premises of the rules. However, the purpose of defining the predicate is to be able to define the unification algorithm, which means that the unification algorithm is not defined yet, and hence it cannot appear as part of the premises of the rules of the special-purpose predicate. We would have the same problem, for example, if we try to use our method to define a predicate that allows us to formalise the Ackermann function in type theory.

Thus, we would like to generalise this method so that it can be used for formalising as many total and general recursive algorithms as possible.

The second research direction that we are interested in pursuing is related to the formalisation of the theory of programming languages in type theory, in particular the theory of functional programming languages. In [Mil78], Milner presents the type inference algorithm $\mathcal{W}$ which is used for inferring the type of expressions in the language ML [MTHM97]. Given an expression e that has type under a context Γ, the algorithm $\mathcal{W}$ gives the *most general type scheme* for e from which all types that can be derived for the expression under Γ are *instances*. The algorithm $\mathcal{W}$, which was proven to be sound and complete in [Dam85], is based on the existence of a unification algorithm with the same properties as the algorithm we present in this work. Thus, we plan to use our formalisation of the unification algorithm to formalise the type inference algorithm $\mathcal{W}$ and the proofs that the algorithm is sound and complete.

Appendix A

ALF Formalisation of the Inequalities over Lists of Pairs of Terms

Here, we explain the ALF proofs of the inequalities over lists of pairs of terms presented in section 4.2.

The functions $\#\mathsf{vars}_{\mathrm{LPT}}$, $\#\mathsf{funs}_{\mathrm{LPT}}$, $\#\mathsf{eqs}_{\mathrm{LPT}}$ and $\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}$ are defined in ALF as follows:

$\#\mathsf{vars}_{\mathrm{LPT}} \in (lp \in \mathrm{ListPT})\ \mathrm{N}$
$\quad \#\mathsf{vars}_{\mathrm{LPT}}(lp) \equiv \mathsf{len}(\mathsf{vars}_{\mathrm{LPT}}(lp))$
$\#\mathsf{funs}_{\mathrm{LPT}} \in (lp \in \mathrm{ListPT})\ \mathrm{N}$
$\quad \#\mathsf{funs}_{\mathrm{LPT}}([]) \equiv 0$
$\quad \#\mathsf{funs}_{\mathrm{LPT}}(:(lp', .(t_1, t_2))) \equiv +(\#\mathsf{funs}_{\mathrm{LPT}}(lp'), +(\#\mathsf{funs}_{\mathrm{T}}(t_1), \#\mathsf{funs}_{\mathrm{T}}(t_2)))$
$\#\mathsf{eqs}_{\mathrm{LPT}} \in (lp \in \mathrm{ListPT})\ \mathrm{N}$
$\quad \#\mathsf{eqs}_{\mathrm{LPT}}([]) \equiv 0$
$\quad \#\mathsf{eqs}_{\mathrm{LPT}}(:(lp', .(\mathsf{var}(x), \mathsf{var}(x_1)))) \equiv \mathrm{Var}_{\mathrm{to}}\mathrm{A}(+(\#\mathsf{eqs}_{\mathrm{LPT}}(lp'), 1), \#\mathsf{eqs}_{\mathrm{LPT}}(lp'), \mathrm{Var}_{\mathrm{dec}}(x, x_1))$
$\quad \#\mathsf{eqs}_{\mathrm{LPT}}(:(lp', .(\mathsf{var}(x), \mathsf{fun}(f, lt)))) \equiv \#\mathsf{eqs}_{\mathrm{LPT}}(lp')$
$\quad \#\mathsf{eqs}_{\mathrm{LPT}}(:(lp', .(\mathsf{fun}(f, lt), \mathsf{var}(x)))) \equiv +(\#\mathsf{eqs}_{\mathrm{LPT}}(lp'), 1)$
$\quad \#\mathsf{eqs}_{\mathrm{LPT}}(:(lp', .(\mathsf{fun}(f, lt), \mathsf{fun}(f_1, lt_1)))) \equiv \#\mathsf{eqs}_{\mathrm{LPT}}(lp')$
$\mathrm{LPT}_{\mathrm{to}}\mathrm{N3} \in (lp \in \mathrm{ListPT})\ \mathrm{N3}$
$\quad \mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(lp) \equiv .(\#\mathsf{vars}_{\mathrm{LPT}}(lp), \#\mathsf{funs}_{\mathrm{LPT}}(lp), \#\mathsf{eqs}_{\mathrm{LPT}}(lp))$

The function len is not the conventional function length over lists because it does not count the actual number of variables in a list but the number of different variables in the list. This cumbersome solution is due to the fact that we cannot directly formalise the notion of (mathematical) sets in ALF. The function $\#\mathsf{funs}_{\mathrm{T}}$ counts the number of function applications that occur in a term. The corresponding function for vectors of terms is called $\#\mathsf{funs}_{\mathrm{VT}}$. The ALF definitions of these two mutually recursive functions are given in section C.4.1.

The ALF lemmas corresponding to the four inequalities over lists of pairs of terms presented in section 4.2 are the following:

$<_{\mathrm{LPTvar\_var}} \in (x \in \mathrm{Var};\ lp \in \mathrm{ListPT}) <_{\mathrm{N3}}(\mathrm{LPT_{to}N3}(lp), \mathrm{LPT_{to}N3}(:(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$

$<_{\mathrm{LPT}:=_{\mathrm{var\_term}}} \in (lp \in \mathrm{ListPT};\ \notin_{\mathrm{L}}(x, \mathrm{vars_T}(t))) <_{\mathrm{N3}}(\mathrm{LPT_{to}N3}(:=_{\mathrm{LPT}}(x, t, lp)), \mathrm{LPT_{to}N3}(:(lp, .(\mathrm{var}(x), t))))$

$<_{\mathrm{LPTvar\_fun}} \in (f \in \mathrm{Fun};\ x \in \mathrm{Var};\ lt \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) <_{\mathrm{N3}}(\mathrm{LPT_{to}N3}(:(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt)))), \mathrm{LPT_{to}N3}(:(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$

$<_{\mathrm{LPTzip\_fun\_fun}} \in (f, g \in \mathrm{Fun};\ lt_1, lt_2 \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) <_{\mathrm{N3}}(\mathrm{LPT_{to}N3}(++(\mathrm{zip}(lt_1, lt_2), lp)), \mathrm{LPT_{to}N3}(:(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(g, lt_2)))))$

To prove these lemmas we need three kinds of auxiliary lemmas: lemmas about the number of variables in the lists of pairs of terms, about the number of function applications in the lists and about the number of pairs counted by the function $\#\mathrm{eqs}_{\mathrm{LPT}}$. Below, we describe the main auxiliary lemmas needed in order to prove these four inequalities over lists of pairs of terms. See section C.4.4 for the complete ALF proofs of these inequalities.

Lemmas about the Number of Variables: We prove the following four lemmas about the number of variables in a list of pairs of terms:

$\leq\#\mathrm{vars}_{\mathrm{var\_var}} \in (x \in \mathrm{Var};\ lp \in \mathrm{ListPT}) \leq(\#\mathrm{vars}_{\mathrm{LPT}}(lp), \#\mathrm{vars}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$

$<\#\mathrm{vars}_{:=_{\mathrm{LPT}}} \in (lp \in \mathrm{ListPT};\ \notin_{\mathrm{L}}(x, \mathrm{vars_T}(t))) <(\#\mathrm{vars}_{\mathrm{LPT}}(:=_{\mathrm{LPT}}(x, t, lp)), \#\mathrm{vars}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), t))))$

$=\#\mathrm{vars}_{\mathrm{var\_fun}} \in (f \in \mathrm{Fun};\ x \in \mathrm{Var};\ lt \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) =(\#\mathrm{vars}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt)))), \#\mathrm{vars}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$

$=\#\mathrm{vars}_{\mathrm{fun\_fun}} \in (f, g \in \mathrm{Fun};\ lt_1, lt_2 \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) =(\#\mathrm{vars}_{\mathrm{LPT}}(++(\mathrm{zip}(lt_1, lt_2), lp)), \#\mathrm{vars}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(g, lt_2)))))$

To prove the above lemmas, we use the following auxiliary lemmas[1]:

$\subset_{\mathrm{to}}< \in (l_1 \in \mathrm{ListVar};\ \neg(\in_{\mathrm{L}}(x, l_1));\ \in_{\mathrm{L}}(x, l_2);\ \subseteq(l_1, l_2)) <(\mathrm{len}(l_1), \mathrm{len}(l_2))$

$\subseteq_{\mathrm{to}}\leq \in (\subseteq(l_1, l_2)) \leq(\mathrm{len}(l_1), \mathrm{len}(l_2))$

Given the lists of variables l_1 and l_2, to prove that $\mathrm{len}(l_1) < \mathrm{len}(l_2)$ we have to prove that l_1 is included in l_2 and we have to find a variable x in l_2 that does not belong to the list l_1, and to prove that $\mathrm{len}(l_1) \leq \mathrm{len}(l_2)$ it is sufficient to prove that l_1 is included in l_2. Finally, to prove that $\mathrm{len}(l_1) = \mathrm{len}(l_2)$ we prove that l_1 is included in l_2 and that l_2 is included in l_1. This, in turn, gives us proofs that $\mathrm{len}(l_1) \leq \mathrm{len}(l_2)$ and that $\mathrm{len}(l_2) \leq \mathrm{len}(l_1)$, which clearly gives us a proof that $\mathrm{len}(l_1) = \mathrm{len}(l_2)$.

[1] See section C.3.3 for the complete proof of these two lemmas.

In [Rou92], a similar technique is used to prove inequalities about the numbers of variables in a list of variables.

Lemmas about the Number of Functions Applications: We prove the following three lemmas about the number of functions applications in a list of pairs of terms:

$$=\#\mathrm{funs}_{\mathrm{var\_var}} \in (x \in \mathrm{Var};\ lp \in \mathrm{ListPT}) =(\#\mathrm{funs}_{\mathrm{LPT}}(lp), \#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$$

$$=\#\mathrm{funs}_{\mathrm{var\_fun}} \in (f \in \mathrm{Fun};\ x \in \mathrm{Var};\ lt \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) =(\#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt)))), \#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$$

$$<\#\mathrm{funs}_{\mathrm{fun\_fun}} \in (f, g \in \mathrm{Fun};\ lt_1, lt_2 \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) <(\#\mathrm{funs}_{\mathrm{LPT}}(++(\mathrm{zip}(lt_1, lt_2), lp)), \#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(g, lt_2)))))$$

To prove the last lemma, we use the following two extra auxiliary lemmas:

$$=\#\mathrm{funs}_{\mathrm{zip}} \in (lt_1, lt_2 \in \mathrm{VTerm}(n)) =(\#\mathrm{funs}_{\mathrm{LPT}}(\mathrm{zip}(lt_1, lt_2)), +(\#\mathrm{funs}_{\mathrm{VT}}(lt_1), \#\mathrm{funs}_{\mathrm{VT}}(lt_2)))$$

$$=\#\mathrm{funs}_{++} \in (lp_1, lp_2 \in \mathrm{ListPT}) =(\#\mathrm{funs}_{\mathrm{LPT}}(++(lp_1, lp_2)), +(\#\mathrm{funs}_{\mathrm{LPT}}(lp_1), \#\mathrm{funs}_{\mathrm{LPT}}(lp_2)))$$

The proofs of these five lemmas are straightforward. In them, we mainly use the definition of the function $\#\mathrm{funs}_{\mathrm{LPT}}$ and the associative and commutative properties of the addition of natural numbers.

Lemmas about the Number of Pairs Counted by the Function $\#\mathrm{eqs}_{\mathrm{LPT}}$: We prove the following two lemmas about the number of pairs in a list of pairs of terms counted by the function $\#\mathrm{eqs}_{\mathrm{LPT}}$:

$$<\#\mathrm{eqs}_{\mathrm{var\_var}} \in (x \in \mathrm{Var};\ lp \in \mathrm{ListPT}) <(\#\mathrm{eqs}_{\mathrm{LPT}}(lp), \#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$$

$$<\#\mathrm{eqs}_{\mathrm{var\_fun}} \in (f \in \mathrm{Fun};\ x \in \mathrm{Var};\ lt \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) <(\#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt)))), \#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$$

The proofs of these lemmas are straightforward and they mainly use the definition of the function $\#\mathrm{eqs}_{\mathrm{LPT}}$.

Appendix B

All Lists of Pairs of Terms Satisfy UniAcc

In this appendix we discuss the proof that shows that all lists of pairs of terms satisfy the predicate UniAcc.

For this purpose, we define a function $\mathrm{P_n}$ over triples of natural numbers. Given $n_3 \in \mathrm{N}^3$, we define $\mathrm{P_n}$ as follows:

$$\mathrm{P_n}(n_3) \in \forall\ lp' \in \mathrm{ListPT}\,.\,(\mathrm{LPT_{to}N3}(lp') = n_3) \Rightarrow \mathrm{UniAcc}(lp')$$

Given a list $lp \in \mathrm{ListPT}$, we have that $\mathrm{LPT_{to}N3}(lp) \in \mathrm{N}^3$ and then we obtain that

$$\mathrm{P_n}(\mathrm{LPT_{to}N3}(lp)) \in \forall\ lp' \in \mathrm{ListPT}\,.\,(\mathrm{LPT_{to}N3}(lp') = \mathrm{LPT_{to}N3}(lp)) \Rightarrow \mathrm{UniAcc}(lp')$$

So, if we perform one $\forall$_elimination and one $\Rightarrow$_elimination, the former with the list lp and the latter with a proof that $\mathrm{LPT_{to}N3}(lp) = \mathrm{LPT_{to}N3}(lp)$, we obtain a proof that $\mathrm{UniAcc}(lp)$. Then, we have the following ALF lemma:

$\mathrm{allUniAcc_{LPT}} \in (lp \in \mathrm{ListPT})\ \mathrm{UniAcc}(lp)$
$\mathrm{allUniAcc_{LPT}}(lp) \equiv \Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(\mathrm{P_n}(\mathrm{LPT_{to}N3}(lp)), lp), \mathrm{refl}(\mathrm{LPT_{to}N3}(lp)))$

To prove the predicate $\mathrm{P_n}$ we use the fact that N^3 is well-founded. Given a lemma $\mathrm{uniacc_{aux}}$ of type:

$\mathrm{uniacc_{aux}} \in (\mathrm{Acc}(\mathrm{N3}, <_{\mathrm{N3}}, n_3);$
$f \in (m_3 \in \mathrm{N3}; <_{\mathrm{N3}}(m_3, n_3))\ \forall(\mathrm{ListPT}, [lp]\Rightarrow(=(\mathrm{LPT_{to}N3}(lp), m_3), \mathrm{UniAcc}(lp)))$
$)\ \forall(\mathrm{ListPT}, [lp]\Rightarrow(=(\mathrm{LPT_{to}N3}(lp), n_3), \mathrm{UniAcc}(lp)))$

we can use the rule of well-founded recursion to construct a proof of $\mathrm{P_n}$. Hence, we have that:

$\mathrm{P_n} \in (n_3 \in \mathrm{N3})\ \forall(\mathrm{ListPT}, [lp]\Rightarrow(=(\mathrm{LPT_{to}N3}(lp), n_3), \mathrm{UniAcc}(lp)))$
$\mathrm{P_n}(n_3) \equiv \mathrm{wfrec}(n_3, \mathrm{allacc_{N3}}(n_3), \mathrm{uniacc_{aux}})$

The lemma $\mathrm{uniacc_{aux}}$ is defined in ALF as follows:

$$\begin{aligned}
&\mathrm{uniacc}_{aux} \in (\mathrm{Acc}(\mathrm{N3}, <_{\mathrm{N3}}, n_3);\\
&\quad f \in (m_3 \in \mathrm{N3}; <_{\mathrm{N3}}(m_3, n_3)) \forall(\mathrm{ListPT}, [lp]\Rightarrow(=(\mathrm{LPT}_{to}\mathrm{N3}(lp), m_3), \mathrm{UniAcc}(lp)))\\
&\quad)\forall(\mathrm{ListPT}, [lp]\Rightarrow(=(\mathrm{LPT}_{to}\mathrm{N3}(lp), n_3), \mathrm{UniAcc}(lp)))\\
&\mathrm{uniacc}_{aux}(h, f) \equiv \forall_I([lp]\Rightarrow_I([h']\mathrm{uniacc}_{aux2}(h, f, lp, h')))
\end{aligned}$$

where the lemma uniacc_{aux2} has the following type:

$$\begin{aligned}
&\mathrm{uniacc}_{aux2} \in (\mathrm{Acc}(\mathrm{N3}, <_{\mathrm{N3}}, n_3);\\
&\quad f \in (m_3 \in \mathrm{N3}; <_{\mathrm{N3}}(m_3, n_3)) \forall(\mathrm{ListPT}, [lp']\Rightarrow(=(\mathrm{LPT}_{to}\mathrm{N3}(lp'), m_3), \mathrm{UniAcc}(lp')));\\
&\quad lp \in \mathrm{ListPT};\\
&\quad =(\mathrm{LPT}_{to}\mathrm{N3}(lp), n_3)\\
&\quad)\,\mathrm{UniAcc}(lp)
\end{aligned}$$

To prove this last lemma, we proceed in a similar way as the one presented in section 4.4.1 to define the algorithm Unify_{acc}: we perform a few pattern matchings on the list lp and a few case analyses using the same decidability lemmas as the ones used when defining Unify_{acc}. In this way, we obtain an incomplete ALF proof consisting of nine cases where, for each case, we have to supply a proof that the corresponding list satisfies UniAcc. To build each of these proofs we use the UniAcc constructor that corresponds to the list in turn. We present the complete ALF proof of lemma uniacc_{aux2} in figure B.1. Notice that in the proof of this lemma we again make use of the inequality lemmas presented in appendix A. Finally, notice that the ALF code of the formalisation of this proof and the ALF code of the algorithm Unify_{acc} have similar skeletons.

```
uniacc_aux2 ∈ (Acc(N3, <_N3, n_3);
               f ∈ (m_3 ∈ N3; <_N3(m_3, n_3)) ∀(ListPT, [lp']⇒(=(LPT_toN3(lp'), m_3), UniAcc(lp')));
               lp ∈ ListPT;
               =(LPT_toN3(lp), n_3)
              ) UniAcc(lp)
  uniacc_aux2(p, f, [], h) ≡ uniacc[]
  uniacc_aux2(p, f, :(lp_1, .(var(x), var(x_1))), refl(_)) ≡
    case Var_dec(x, x_1) ∈ Dec(=(x, x_1)) of
      yes(refl(_)) ⇒
        uniacc_var_var(x_1, ⇒_E(∀_E(f(LPT_toN3(lp_1), <_LPTvar_var(x_1, lp_1)), lp_1), refl(LPT_toN3(lp_1))))
      no(h) ⇒
        uniacc:=_var_term(∉:(∉[](x), h),
                          ⇒_E(∀_E(f(LPT_toN3(:=_LPT(x, var(x_1)), lp_1)), <_LPT:=_var_term(lp_1, ∉:(∉[](x), h))),
                              :=_LPT(x, var(x_1)), lp_1)),
                              refl(LPT_toN3(:=_LPT(x, var(x_1)), lp_1)))))
    end
  uniacc_aux2(p, f, :(lp_1, .(var(x), fun(f_1, lt_1))), refl(_)) ≡
    case ∈_dec(x, vars_VT(lt_1)) ∈ Dec(∈_L(x, vars_VT(lt_1))) of
      yes(h) ⇒ uniacc_var_term(lp_1, h, ≠_T(f_1, x, lt_1))
      no(h) ⇒
        uniacc:=_var_term(¬∈_to∉(vars_T(fun(f_1, lt_1)), h),
                          ⇒_E(∀_E(f(LPT_toN3(:=_LPT(x, fun(f_1, lt_1), lp_1)),
                                  <_LPT:=_var_term(lp_1, ¬∈_to∉(vars_T(fun(f_1, lt_1)), h))),
                                  :=_LPT(x, fun(f_1, lt_1), lp_1)),
                                  refl(LPT_toN3(:=_LPT(x, fun(f_1, lt_1), lp_1)))))
    end
  uniacc_aux2(p, f, :(lp_1, .(fun(f_1, lt_1), var(x))), refl(_)) ≡
    uniacc_var_fun(⇒_E(∀_E(f(LPT_toN3(:(lp_1, .(var(x), fun(f_1, lt_1)))), <_LPTvar_fun(f_1, x, lt_1, lp_1)),
                           :(lp_1, .(var(x), fun(f_1, lt_1)))),
                       refl(LPT_toN3(:(lp_1, .(var(x), fun(f_1, lt_1)))))))
  uniacc_aux2(p, f, :(lp_1, .(fun(f_1, lt_1), fun(f_2, lt_2))), refl(_)) ≡
    case Fun_dec(f_1, f_2) ∈ Dec(=(f_1, f_2)) of
      yes(refl(_)) ⇒
        case N_dec(n_1, n_2) ∈ Dec(=(n_1, n_2)) of
          yes(refl(_)) ⇒
            uniacc_zip_fun_fun(
              f_2,
              ⇒_E(∀_E(f(LPT_toN3(++(zip(lt_1, lt_2), lp_1)), <_LPTzip_fun_fun(f_2, f_2, lt_1, lt_2, lp_1)),
                      ++(zip(lt_1, lt_2), lp_1)),
                      refl(LPT_toN3(++(zip(lt_1, lt_2), lp_1)))))
          no(h_1) ⇒ uniacc_fun_fun(lt_1, lt_2, lp_1, ∨_R(h_1))
        end
      no(h) ⇒ uniacc_fun_fun(lt_1, lt_2, lp_1, ∨_L(h))
    end
```

Figure B.1: ALF Proof of lemma $\text{uniacc}_{\text{aux2}}$

Appendix C

ALF Codes

This appendix contains the complete ALF code of the formalisation of the unification algorithm in Martin-Löf's type theory.

C.1 Logical Constants

C.1.1 Absurdity

$\bot \in$ **Set**

C.1.2 Conjunction

$\wedge \in (A, B \in$ **Set**$)$ **Set**
  $\wedge_{\mathrm{I}} \in (a \in A;\ b \in B)\ \wedge(A, B)$
$\wedge_3 \in (A, B, C \in$ **Set**$)$ **Set**
  $\wedge_{\mathrm{I}3} \in (a \in A;\ b \in B;\ c \in C)\ \wedge_3(A, B, C)$
fst $\in (\wedge(A, B))\ A$
  fst$(\wedge_{\mathrm{I}}(a, b)) \equiv a$
snd $\in (\wedge(A, B))\ B$
  snd$(\wedge_{\mathrm{I}}(a, b)) \equiv b$

C.1.3 Disjunction

$\vee \in (A, B \in$ **Set**$)$ **Set**
  $\vee_{\mathrm{L}} \in (a \in A)\ \vee(A, B)$
  $\vee_{\mathrm{R}} \in (b \in B)\ \vee(A, B)$

C.1.4 Equivalence

$\Leftrightarrow \in (A, B \in$ **Set**$)$ **Set**
  $\Leftrightarrow \equiv [A, B]\wedge(\Rightarrow(A, B), \Rightarrow(B, A))$

C.1.5 Existential Quantifier

$\exists \in (A \in \mathbf{Set};\ B \in (A)\,\mathbf{Set})\ \mathbf{Set}$
$\quad \exists_{\mathrm{I}} \in (a \in A;\ b \in B(a))\ \exists(A, B)$
$\mathrm{witness} \in (\exists(A, B))\ A$
$\quad \mathrm{witness}(\exists_{\mathrm{I}}(a, b)) \equiv a$
$\mathrm{proof} \in (h \in \exists(A, B))\ B(\mathrm{witness}(h))$
$\quad \mathrm{proof}(\exists_{\mathrm{I}}(a, b)) \equiv b$

C.1.6 Implication

$\Rightarrow \in (A, B \in \mathbf{Set})\ \mathbf{Set}$
$\quad \Rightarrow_{\mathrm{I}} \in (f \in (A)\,B)\ \Rightarrow(A, B)$
$\Rightarrow_{\mathrm{E}} \in (f \in \Rightarrow(A, B);\ a \in A)\ B$
$\quad \Rightarrow_{\mathrm{E}}(\Rightarrow_{\mathrm{I}}(f_1), a) \equiv f_1(a)$
$\Rightarrow_{\mathrm{trans}} \in (\Rightarrow(A, B);\ \Rightarrow(B, C))\ \Rightarrow(A, C)$
$\quad \Rightarrow_{\mathrm{trans}}(\Rightarrow_{\mathrm{I}}(f), \Rightarrow_{\mathrm{I}}(f_1)) \equiv \Rightarrow_{\mathrm{I}}([a]f_1(f(a)))$

C.1.7 Negation

$\neg \in (A \in \mathbf{Set})\ \mathbf{Set}$
$\quad \neg \equiv [A]\Rightarrow(A, \bot)$

C.1.8 Universal Quantifier

$\forall \in (A \in \mathbf{Set};\ B \in (A)\,\mathbf{Set})\ \mathbf{Set}$
$\quad \forall_{\mathrm{I}} \in (f \in (a \in A)\,B(a))\ \forall(A, B)$
$\forall_{\mathrm{E}} \in (\forall(A, B);\ a \in A)\ B(a)$
$\quad \forall_{\mathrm{E}}(\forall_{\mathrm{I}}(f), a) \equiv f(a)$

C.2 Some Useful General Predicates

C.2.1 Accessibility

$\mathrm{Acc} \in (A \in \mathbf{Set};\ less \in (A; A)\,\mathbf{Set};\ a \in A)\ \mathbf{Set}$
$\quad \mathrm{acc} \in (a \in A;\ (x \in A;\ less(x, a))\,\mathrm{Acc}(A, less, x))\ \mathrm{Acc}(A, less, a)$
$\mathrm{WF} \in (A \in \mathbf{Set};\ less \in (A; A)\,\mathbf{Set})\ \mathbf{Set}$
$\quad \mathrm{WF} \equiv [A, less]\forall(A, [a]\mathrm{Acc}(A, less, a))$
$\mathrm{wfrec} \in (a \in A;$
$\qquad \mathrm{Acc}(A, less, a);$
$\qquad e \in (x \in A;\ \mathrm{Acc}(A, less, x);\ (y \in A; less(y, x))P(y))\,P(x)$
$\qquad)\,P(a)$
$\quad \mathrm{wfrec}(a, \mathrm{acc}(\_, p), e) \equiv e(a, \mathrm{acc}(a, p), [y, h]\mathrm{wfrec}(y, p(y, h), e))$

C.2.2 Decidability

$\mathrm{Dec} \in (\mathbf{Set})\ \mathbf{Set}$
$\quad \mathrm{yes} \in (h \in P)\ \mathrm{Dec}(P)$
$\quad \mathrm{no} \in (h \in \neg(P))\ \mathrm{Dec}(P)$

C.2.3 Propositional Equality

$= \in (a, b \in A)$ **Set**
  refl $\in (a \in A) =(a, a)$
$=_{symm} \in (=(a, b)) =(b, a)$
  $=_{symm}(\text{refl}(\_)) \equiv \text{refl}(b)$
$=_{trans} \in (=(a, b); =(b, c)) =(a, c)$
  $=_{trans}(\text{refl}(\_), \text{refl}(\_)) \equiv \text{refl}(c)$
$=_{cong1} \in (f \in (A)\, B; =(a_1, a_2)) =(f(a_1), f(a_2))$
  $=_{cong1}(f, \text{refl}(\_)) \equiv \text{refl}(f(a_2))$
$=_{cong2} \in (f \in (A; B)\, C; =(a_1, a_2); =(b_1, b_2)) =(f(a_1, b_1), f(a_2, b_2))$
  $=_{cong2}(f, \text{refl}(\_), \text{refl}(\_)) \equiv \text{refl}(f(a_2, b_2))$
$=_{subst1} \in (=(a, b); P(a))\, P(b)$
  $=_{subst1}(\text{refl}(\_), h_1) \equiv h_1$
$=_{subst2} \in (=(a, b); =(c, d); R(b, c))\, R(a, d)$
  $=_{subst2}(\text{refl}(\_), \text{refl}(\_), h_2) \equiv h_2$

C.3 Some Useful General Data Types

C.3.1 Error

Error $\in$ **Set**
  error $\in$ Error

C.3.2 Lists

List $\in (A \in$ **Set**$)$ **Set**
  [] $\in$ List(A)
  : $\in (l \in \text{List}(A); a \in A)$ List(A)
++ $\in (l_1, l_2 \in \text{List}(A))$ List(A)
  $++(l_1, []) \equiv l_1$
  $++(l_1, :(l, a)) \equiv :(++(l_1, l), a)$
$\in_L \in (a \in A; l \in \text{List}(A))$ **Set**
  $\in_{hd} \in (a \in A; l \in \text{List}(A)) \in_L(a, :(l, a))$
  $\in_{tl} \in (b \in A; l \in \text{List}(A); \in_L(a, l)) \in_L(a, :(l, b))$
$\notin_L \in (a \in A; l \in \text{List}(A))$ **Set**
  $\notin_{[]} \in (a \in A) \notin_L(a, [])$
  $\notin_{:} \in (\notin_L(a, l); \neg(=(a, b))) \notin_L(a, :(l, b))$
Disjoint $\in (l, l_1 \in \text{List}(A))$ **Set**
  $\text{disj}_{[]} \in (l_1 \in \text{List}(A))\ \text{Disjoint}([], l_1)$
  $\text{disj}_{:} \in (\text{Disjoint}(l, l_1); \notin_L(a, l_1))\ \text{Disjoint}(:(l, a), l_1)$
$\subseteq \in (l_1, l_2 \in \text{List}(A))$ **Set**
  $\subseteq_{[]} \in (l \in \text{List}(A)) \subseteq([], l)$
  $\subseteq_{:} \in (\subseteq(l_1, l_2); \in_L(a, l_2)) \subseteq(:(l_1, a), l_2)$
$=_{list^{to}=_{head}} \in (=(:(l_1, a), :(l_2, b))) =(a, b)$
  $=_{list^{to}=_{head}}(\text{refl}(\_)) \equiv \text{refl}(b)$
$=_{list^{to}=_{tail}} \in (=(:(l_1, a), :(l_2, b))) =(l_1, l_2)$
  $=_{list^{to}=_{tail}}(\text{refl}(\_)) \equiv \text{refl}(l_2)$

$\in_{:\mathrm{red}} \in (\neg(=(a, b)); \in_{\mathrm{L}}(a, :(l, b))) \in_{\mathrm{L}}(a, l)$

$\in_{:\mathrm{red}}(\Rightarrow_{\mathrm{I}}(f), \in_{\mathrm{hd}}(\_, \_)) \equiv$ **case** $f(\mathrm{refl}(b)) \in \bot$ **of**
end

$\in_{:\mathrm{red}}(h, \in_{\mathrm{tl}}(\_, \_, h_2)) \equiv h_2$

$\in_{[]_{\mathrm{to}}\bot} \in (\in_{\mathrm{L}}(a, [])) \bot$

$\in_{[]_{\mathrm{to}}\bot}(h) \equiv$ **case** $h \in \in_{\mathrm{L}}(a, [])$ **of**
end

$\in_{:\mathrm{to}\bot} \in (\neg(=(a, b)); \neg(\in_{\mathrm{L}}(a, l)); \in_{\mathrm{L}}(a, :(l, b))) \bot$

$\in_{:\mathrm{to}\bot}(\Rightarrow_{\mathrm{I}}(f), h_1, \in_{\mathrm{hd}}(\_, \_)) \equiv f(\mathrm{refl}(b))$

$\in_{:\mathrm{to}\bot}(h, \Rightarrow_{\mathrm{I}}(f), \in_{\mathrm{tl}}(\_, \_, h_3)) \equiv f(h_3)$

$\in{+}{+}_{\mathrm{monR}} \in (l_2 \in \mathrm{List}(A); \in_{\mathrm{L}}(a, l_1)) \in_{\mathrm{L}}(a, {+}{+}(l_1, l_2))$

$\in{+}{+}_{\mathrm{monR}}([], h) \equiv h$

$\in{+}{+}_{\mathrm{monR}}(:(l, a_1), h) \equiv \in_{\mathrm{tl}}(a_1, {+}{+}(l_1, l), \in{+}{+}_{\mathrm{monR}}(l, h))$

$\in{+}{+}_{\mathrm{monL}} \in (l_1 \in \mathrm{List}(A); \in_{\mathrm{L}}(a, l_2)) \in_{\mathrm{L}}(a, {+}{+}(l_1, l_2))$

$\in{+}{+}_{\mathrm{monL}}(l_1, \in_{\mathrm{hd}}(\_, l)) \equiv \in_{\mathrm{hd}}(a, {+}{+}(l_1, l))$

$\in{+}{+}_{\mathrm{monL}}(l_1, \in_{\mathrm{tl}}(b, l, h_1)) \equiv \in_{\mathrm{tl}}(b, {+}{+}(l_1, l), \in{+}{+}_{\mathrm{monL}}(l_1, h_1))$

$\in{+}{+}_{\mathrm{to}}\in\vee \in (l_2 \in \mathrm{List}(A); \in_{\mathrm{L}}(a, {+}{+}(l_1, l_2))) \vee(\in_{\mathrm{L}}(a, l_1), \in_{\mathrm{L}}(a, l_2))$

$\in{+}{+}_{\mathrm{to}}\in\vee([], h) \equiv \vee_{\mathrm{L}}(h)$

$\in{+}{+}_{\mathrm{to}}\in\vee(:(l, a_1), \in_{\mathrm{hd}}(\_, \_)) \equiv \vee_{\mathrm{R}}(\in_{\mathrm{hd}}(a_1, l))$

$\in{+}{+}_{\mathrm{to}}\in\vee(:(l, a_1), \in_{\mathrm{tl}}(\_, \_, h_1)) \equiv$ **case** $\in{+}{+}_{\mathrm{to}}\in\vee(l, h_1) \in \vee(\in_{\mathrm{L}}(a, l_1), \in_{\mathrm{L}}(a, l))$ **of**
$\vee_{\mathrm{L}}(h) \Rightarrow \vee_{\mathrm{L}}(h)$
$\vee_{\mathrm{R}}(h) \Rightarrow \vee_{\mathrm{R}}(\in_{\mathrm{tl}}(a_1, l, h))$
end

$\in{\wedge}\mathrm{disj}_{\mathrm{to}}\notin \in (\in_{\mathrm{L}}(a, l_1); \mathrm{Disjoint}(l_1, l_2)) \notin_{\mathrm{L}}(a, l_2)$

$\in{\wedge}\mathrm{disj}_{\mathrm{to}}\notin(\in_{\mathrm{hd}}(\_, l), \mathrm{disj}_{:}(h, h_2)) \equiv h_2$

$\in{\wedge}\mathrm{disj}_{\mathrm{to}}\notin(\in_{\mathrm{tl}}(b, l, h_2), \mathrm{disj}_{:}(h, h_3)) \equiv \in{\wedge}\mathrm{disj}_{\mathrm{to}}\notin(h_2, h)$

$\notin_{\mathrm{to}}\neq \in (\notin_{\mathrm{L}}(a, :([], b))) \neg(=(a, b))$

$\notin_{\mathrm{to}}\neq(\notin_{:}(h_1, h_2)) \equiv h_2$

$\notin{+}{+} \in (\notin_{\mathrm{L}}(a, l_1); \notin_{\mathrm{L}}(a, l_2)) \notin_{\mathrm{L}}(a, {+}{+}(l_1, l_2))$

$\notin{+}{+}(h, \notin_{[]}(\_)) \equiv h$

$\notin{+}{+}(h, \notin_{:}(h_2, h_3)) \equiv \notin_{:}(\notin{+}{+}(h, h_2), h_3)$

$\notin{+}{+}_{\mathrm{redL}} \in (l_2 \in \mathrm{List}(A); \notin_{\mathrm{L}}(a, {+}{+}(l_1, l_2))) \notin_{\mathrm{L}}(a, l_2)$

$\notin{+}{+}_{\mathrm{redL}}([], h) \equiv \notin_{[]}(a)$

$\notin{+}{+}_{\mathrm{redL}}(:(l, a_1), \notin_{:}(h_1, h_2)) \equiv \notin_{:}(\notin{+}{+}_{\mathrm{redL}}(l, h_1), h_2)$

$\notin{+}{+}_{\mathrm{redR}} \in (l_2 \in \mathrm{List}(A); \notin_{\mathrm{L}}(a, {+}{+}(l_1, l_2))) \notin_{\mathrm{L}}(a, l_1)$

$\notin{+}{+}_{\mathrm{redR}}([], h) \equiv h$

$\notin{+}{+}_{\mathrm{redR}}(:(l, a_1), \notin_{:}(h_1, h_2)) \equiv \notin{+}{+}_{\mathrm{redR}}(l, h_1)$

$\notin_{\mathrm{to}}\neg\in \in (\notin_{\mathrm{L}}(a, l)) \neg(\in_{\mathrm{L}}(a, l))$

$\notin_{\mathrm{to}}\neg\in(\notin_{[]}(\_)) \equiv \Rightarrow_{\mathrm{I}}(\in_{[]_{\mathrm{to}}\bot})$

$\notin_{\mathrm{to}}\neg\in(\notin_{:}(h_1, h_2)) \equiv \Rightarrow_{\mathrm{I}}(\in_{:\mathrm{to}\bot}(h_2, \notin_{\mathrm{to}}\neg\in(h_1)))$

$\neg\in{\wedge}{=}\mathrm{var}_{\mathrm{to}}\bot \in (\neg(\in_{\mathrm{L}}(a, :(l, b))); =(a, b)) \bot$

$\neg\in{\wedge}{=}\mathrm{var}_{\mathrm{to}}\bot(\Rightarrow_{\mathrm{I}}(f), \mathrm{refl}(\_)) \equiv f(\in_{\mathrm{hd}}(b, l))$

$\neg\in_{:\mathrm{to}}\wedge \in (\neg(\in_{\mathrm{L}}(a, :(l, b)))) \wedge(\neg(=(a, b)), \neg(\in_{\mathrm{L}}(a, l)))$

$\neg\in_{:\mathrm{to}}\wedge(\Rightarrow_{\mathrm{I}}(f)) \equiv \wedge_{\mathrm{I}}(\Rightarrow_{\mathrm{I}}([h]\neg\in{\wedge}{=}\mathrm{var}_{\mathrm{to}}\bot(\Rightarrow_{\mathrm{I}}(f), h)), \Rightarrow_{\mathrm{I}}([h]f(\in_{\mathrm{tl}}(b, l, h))))$

$\neg\in_{\mathrm{to}}\notin \in (l \in \mathrm{List}(A); \neg(\in_{\mathrm{L}}(a, l))) \notin_{\mathrm{L}}(a, l)$

$\neg\in_{\mathrm{to}}\notin([], h) \equiv \notin_{[]}(a)$

$\neg\in_{\mathrm{to}}\notin(:(l_1, a_1), h) \equiv$ **case** $\neg\in_{:\mathrm{to}}\wedge(h) \in \wedge(\neg(=(a, a_1)), \neg(\in_{\mathrm{L}}(a, l_1)))$ **of**
$\wedge_{\mathrm{I}}(h_1, h_2) \Rightarrow \notin_{:}(\neg\in_{\mathrm{to}}\notin(l_1, h_2), h_1)$
end

$\in\wedge\notin_{to}\neq$ $\in$ $(\in_L(a, l); \notin_L(b, l)) \neg(=(b, a))$

$\in\wedge\notin_{to}\neq(\in_{hd}(\_, l_1), \notin_:(h, h_2)) \equiv h_2$

$\in\wedge\notin_{to}\neq(\in_{tl}(b_1, l_1, h_2), \notin_:(h, h_3)) \equiv \in\wedge\notin_{to}\neq(h_2, h)$

$\text{disj}_{redR} \in (\text{Disjoint}(l, :(l_1, a))) \text{ Disjoint}(l, l_1)$

$\text{disj}_{redR}(\text{disj}_{[]}(\_)) \equiv \text{disj}_{[]}(l_1)$

$\text{disj}_{redR}(\text{disj}_:(h_1, \notin_:(h, h_3))) \equiv \text{disj}_:(\text{disj}_{redR}(h_1), h)$

$\text{disj}_{[]R} \in (l \in \text{List}(A)) \text{ Disjoint}(l, [])$

$\text{disj}_{[]R}([]) \equiv \text{disj}_{[]}([])$

$\text{disj}_{[]R}(:(l_1, a)) \equiv \text{disj}_:(\text{disj}_{[]R}(l_1), \notin_{[]}(a))$

$\text{disj}_{:R} \in (\text{Disjoint}(l_1, l_2); \notin_L(a, l_1)) \text{ Disjoint}(l_1, :(l_2, a))$

$\text{disj}_{:R}(\text{disj}_{[]}(\_), h_1) \equiv \text{disj}_{[]}(:(l_2, a))$

$\text{disj}_{:R}(\text{disj}_:(h_2, h_3), \notin_:(h, \Rightarrow_I(f))) \equiv \text{disj}_:(\text{disj}_{:R}(h_2, h), \notin_:(h_3, \Rightarrow_I([h_1]f(=_{symm}(h_1)))))$

$\text{disj}_{symm} \in (\text{Disjoint}(l_1, l_2)) \text{ Disjoint}(l_2, l_1)$

$\text{disj}_{symm}(\text{disj}_{[]}(\_)) \equiv \text{disj}_{[]R}(l_2)$

$\text{disj}_{symm}(\text{disj}_:(h_1, h_2)) \equiv \text{disj}_{:R}(\text{disj}_{symm}(h_1), h_2)$

$\text{disj}_{to\notin} \in (\text{Disjoint}(l_1, :(l_2, a))) \notin_L(a, l_1)$

$\text{disj}_{to\notin}(h) \equiv$ **case** $\text{disj}_{symm}(h) \in \text{Disjoint}(:(l_2, a), l_1)$ **of**

$\text{disj}_:(h_1, h_2) \Rightarrow h_2$

end

$\notin_{to}\text{disj} \in (\notin_L(a, l)) \text{ Disjoint}(:([], a), l)$

$\notin_{to}\text{disj}(\notin_{[]}(\_)) \equiv \text{disj}_{[]R}(:([], a))$

$\notin_{to}\text{disj}(\notin_:(h_1, h_2)) \equiv \text{disj}_:(\text{disj}_{[]}(:(l_1, b)), \notin_:(h_1, h_2))$

$\subseteq_{:monR} \in (a \in A; \subseteq(l_1, l_2)) \subseteq(l_1, :(l_2, a))$

$\subseteq_{:monR}(a, \subseteq_{[]}(\_)) \equiv \subseteq_{[]}(:(l_2, a))$

$\subseteq_{:monR}(a, \subseteq_:(h_1, h_2)) \equiv \subseteq_:(\subseteq_{:monR}(a, h_1), \in_{tl}(a, l_2, h_2))$

$\subseteq_{refl} \in (l \in \text{List}(A)) \subseteq(l, l)$

$\subseteq_{refl}([]) \equiv \subseteq_{[]}([])$

$\subseteq_{refl}(:(l_1, a)) \equiv \subseteq_:(\subseteq_{:monR}(a, \subseteq_{refl}(l_1)), \in_{hd}(a, l_1))$

$\subseteq\in_{trans} \in (\subseteq(l_1, l_2); \in_L(a, l_1)) \in_L(a, l_2)$

$\subseteq\in_{trans}(\subseteq_:(h_1, h_2), \in_{hd}(\_, l)) \equiv h_2$

$\subseteq\in_{trans}(\subseteq_:(h_1, h_3), \in_{tl}(b, l, h_2)) \equiv \subseteq\in_{trans}(h_1, h_2)$

$\subseteq_{trans} \in (\subseteq(l_1, l_2); \subseteq(l_2, l_3)) \subseteq(l_1, l_3)$

$\subseteq_{trans}(\subseteq_{[]}(\_), h_1) \equiv \subseteq_{[]}(l_3)$

$\subseteq_{trans}(\subseteq_:(h_2, h_3), h_1) \equiv \subseteq_:(\subseteq_{trans}(h_2, h_1), \subseteq\in_{trans}(h_1, h_3))$

$\subseteq{+}{+}_L \in (\subseteq(l_1, l); \subseteq(l_2, l)) \subseteq(+{+}(l_1, l_2), l)$

$\subseteq{+}{+}_L(h, \subseteq_{[]}(\_)) \equiv h$

$\subseteq{+}{+}_L(h, \subseteq_:(h_2, h_3)) \equiv \subseteq_:(\subseteq{+}{+}_L(h, h_2), h_3)$

$\subseteq{+}{+}_{refl} \in (l \in \text{List}(A)) \subseteq(+{+}(l, l), l)$

$\subseteq{+}{+}_{refl}(l) \equiv \subseteq{+}{+}_L(\subseteq_{refl}(l), \subseteq_{refl}(l))$

$\subseteq{+}{+}_{monR} \in (l \in \text{List}(A); \subseteq(l_1, l_2)) \subseteq(l_1, +{+}(l_2, l))$

$\subseteq{+}{+}_{monR}(l, \subseteq_{[]}(\_)) \equiv \subseteq_{[]}(+{+}(l_2, l))$

$\subseteq{+}{+}_{monR}(l, \subseteq_:(h_1, h_2)) \equiv \subseteq_:(\subseteq{+}{+}_{monR}(l, h_1), \in{+}{+}_{monR}(l, h_2))$

$\subseteq{+}{+}_{monL} \in (l \in \text{List}(A); \subseteq(l_1, l_2)) \subseteq(l_1, +{+}(l, l_2))$

$\subseteq{+}{+}_{monL}(l, \subseteq_{[]}(\_)) \equiv \subseteq_{[]}(+{+}(l, l_2))$

$\subseteq{+}{+}_{monL}(l, \subseteq_:(h_1, h_2)) \equiv \subseteq_:(\subseteq{+}{+}_{monL}(l, h_1), \in{+}{+}_{monL}(l, h_2))$

$\subseteq{+}{+}_C \in (l_1, l_2, l_3 \in \text{List}(A)) \subseteq(+{+}(l_1, l_2), +{+}(+{+}(l_1, l_3), l_2))$

$\subseteq{+}{+}_C(l_1, l_2, l_3) \equiv \subseteq{+}{+}_L(\subseteq{+}{+}_{monR}(l_2, \subseteq{+}{+}_{monR}(l_3, \subseteq_{refl}(l_1))), \subseteq{+}{+}_{monL}(+{+}(l_1, l_3), \subseteq_{refl}(l_2)))$

$\subseteq{+\!+}_{RR} \in (l_1, l_2, l_3 \in \mathrm{List}(A)) \subseteq(l_3, {+\!+}(l_1, {+\!+}(l_2, l_3)))$

$\quad \subseteq{+\!+}_{RR}(l_1, l_2, l_3) \equiv \subseteq{+\!+}_{monL}(l_1, \subseteq{+\!+}_{monL}(l_2, \subseteq_{refl}(l_3)))$

$\subseteq{+\!+}_{LR} \in (l_1, l_2, l_3 \in \mathrm{List}(A)) \subseteq(l_2, {+\!+}(l_1, {+\!+}(l_2, l_3)))$

$\quad \subseteq{+\!+}_{LR}(l_1, l_2, l_3) \equiv \subseteq{+\!+}_{monL}(l_1, \subseteq{+\!+}_{monR}(l_3, \subseteq_{refl}(l_2)))$

$\subseteq_{move} \in (\subseteq(l, :({+\!+}(l_1, l_2), a))) \subseteq(l, {+\!+}(:(l_1, a), l_2))$

$\quad \subseteq_{move}(\subseteq_{[]}(\_)) \equiv \subseteq_{[]}({+\!+}(:(l_1, a), l_2))$

$\quad \subseteq_{move}(\subseteq_{:}(h_1, \in_{hd}(\_, \_))) \equiv \subseteq_{:}(\subseteq_{move}(h_1), \in{+\!+}_{monR}(l_2, \in_{hd}(a, l_1)))$

$\quad \subseteq_{move}(\subseteq_{:}(h_1, \in_{tl}(\_, \_, h))) \equiv$ **case** $\in{+\!+}_{to}{\in}\vee(l_2, h) \in \vee(\in_L(a_1, l_1), \in_L(a_1, l_2))$ **of**

$\qquad \vee_L(h_2) \Rightarrow \subseteq_{:}(\subseteq_{move}(h_1), \in{+\!+}_{monR}(l_2, \in_{tl}(a, l_1, h_2)))$

$\qquad \vee_R(h_2) \Rightarrow \subseteq_{:}(\subseteq_{move}(h_1), \in{+\!+}_{monL}(:(l_1, a), h_2))$

end

$\subseteq_{comm} \in (l_1, l_2 \in \mathrm{List}(A)) \subseteq({+\!+}(l_1, l_2), {+\!+}(l_2, l_1))$

$\quad \subseteq_{comm}(l_1, []) \equiv \subseteq{+\!+}_{monL}([], \subseteq_{refl}(l_1))$

$\quad \subseteq_{comm}(l_1, :(l, a)) \equiv$

$\qquad \subseteq_{trans}(\subseteq_{:}(\subseteq_{trans}(\subseteq_{comm}(l_1, l), \subseteq_{:monR}(a, \subseteq_{refl}({+\!+}(l, l_1)))), \in_{hd}(a, {+\!+}(l, l_1))),$

$\qquad\quad \subseteq_{move}(\subseteq_{refl}(:({+\!+}(l, l_1), a))))$

$\subseteq_{commR} \in (\subseteq(l, {+\!+}(l_1, l_2))) \subseteq(l, {+\!+}(l_2, l_1))$

$\quad \subseteq_{commR}(h) \equiv \subseteq_{trans}(h, \subseteq_{comm}(l_1, l_2))$

$\subseteq{+\!+}_{mon2R} \in (l \in \mathrm{List}(A); \subseteq(l_1, l_2)) \subseteq({+\!+}(l_1, l), {+\!+}(l_2, l))$

$\quad \subseteq{+\!+}_{mon2R}([], h) \equiv h$

$\quad \subseteq{+\!+}_{mon2R}(:(l_3, a), h) \equiv \subseteq_{:}(\subseteq_{:monR}(a, \subseteq{+\!+}_{mon2R}(l_3, h)), \in_{hd}(a, {+\!+}(l_2, l_3)))$

$\subseteq{+\!+}_{mon2L} \in (l \in \mathrm{List}(A); \subseteq(l_1, l_2)) \subseteq({+\!+}(l, l_1), {+\!+}(l, l_2))$

$\quad \subseteq{+\!+}_{mon2L}(l, h) \equiv \subseteq_{trans}(\subseteq_{comm}(l, l_1), \subseteq_{trans}(\subseteq{+\!+}_{mon2R}(l, h), \subseteq_{comm}(l_2, l)))$

$\subseteq{+\!+}_{redR} \in (l_2 \in \mathrm{List}(A); \subseteq({+\!+}(l_1, l_2), l)) \subseteq(l_1, l)$

$\quad \subseteq{+\!+}_{redR}([], h) \equiv h$

$\quad \subseteq{+\!+}_{redR}(:(l_3, a), \subseteq_{:}(h_1, h_2)) \equiv \subseteq{+\!+}_{redR}(l_3, h_1)$

$\subseteq{+\!+}_{redL} \in (\subseteq({+\!+}(l_1, l_2), l)) \subseteq(l_2, l)$

$\quad \subseteq{+\!+}_{redL}(h) \equiv \subseteq{+\!+}_{redR}(l_1, \subseteq_{trans}(\subseteq_{comm}(l_2, l_1), h))$

$\subseteq_{assoc1} \in (l_1, l_2, l_3 \in \mathrm{List}(A)) \subseteq({+\!+}({+\!+}(l_1, l_2), l_3), {+\!+}(l_1, {+\!+}(l_2, l_3)))$

$\quad \subseteq_{assoc1}(l_1, l_2, []) \equiv \subseteq_{refl}({+\!+}(l_1, l_2))$

$\quad \subseteq_{assoc1}(l_1, l_2, :(l, a)) \equiv \subseteq_{:}(\subseteq_{:monR}(a, \subseteq_{assoc1}(l_1, l_2, l)), \in_{hd}(a, {+\!+}(l_1, {+\!+}(l_2, l))))$

$\subseteq_{assoc2} \in (l_1, l_2, l_3 \in \mathrm{List}(A)) \subseteq({+\!+}(l_1, {+\!+}(l_2, l_3)), {+\!+}({+\!+}(l_1, l_2), l_3))$

$\quad \subseteq_{assoc2}(l_1, l_2, []) \equiv \subseteq_{refl}({+\!+}(l_1, l_2))$

$\quad \subseteq_{assoc2}(l_1, l_2, :(l, a)) \equiv \subseteq_{:}(\subseteq_{:monR}(a, \subseteq_{assoc2}(l_1, l_2, l)), \in_{hd}(a, {+\!+}({+\!+}(l_1, l_2), l)))$

$\subseteq_{assoc\_comm} \in (l_1, l_2, l_3, l_4 \in \mathrm{List}(A)$

$\qquad) \subseteq({+\!+}({+\!+}(l_1, l_2), {+\!+}(l_3, l_4)), {+\!+}({+\!+}(l_1, l_3), {+\!+}(l_2, l_4)))$

$\quad \subseteq_{assoc\_comm}(l_1, l_2, l_3, l_4) \equiv$

$\qquad \subseteq_{trans}(\subseteq_{assoc1}(l_1, l_2, {+\!+}(l_3, l_4)),$

$\qquad\quad \subseteq_{trans}(\subseteq{+\!+}_{mon2L}(l_1,$

$\qquad\qquad\qquad \subseteq_{trans}(\subseteq_{assoc2}(l_2, l_3, l_4),$

$\qquad\qquad\qquad\quad \subseteq_{trans}(\subseteq{+\!+}_{mon2R}(l_4, \subseteq_{comm}(l_2, l_3)), \subseteq_{assoc1}(l_3, l_2, l_4)))),$

$\qquad\qquad \subseteq_{assoc2}(l_1, l_3, {+\!+}(l_2, l_4))))$

$\subseteq_{\wedge}\notin_{to}\notin \in (\subseteq(l_1, l_2); \notin_L(a, l_2)) \notin_L(a, l_1)$

$\quad \subseteq_{\wedge}\notin_{to}\notin(\subseteq_{[]}(\_), h_1) \equiv \notin_{[]}(a)$

$\quad \subseteq_{\wedge}\notin_{to}\notin(\subseteq_{:}(h_2, h_3), h_1) \equiv \notin_{:}(\subseteq_{\wedge}\notin_{to}\notin(h_2, h_1), \in_{\wedge}\notin_{to}{\neq}(h_3, h_1))$

$\subseteq_{\wedge}\mathrm{disj}_{to}\mathrm{disj} \in (\subseteq(l_1, l_2); \mathrm{Disjoint}(l_2, l_3))\ \mathrm{Disjoint}(l_1, l_3)$

$\quad \subseteq_{\wedge}\mathrm{disj}_{to}\mathrm{disj}(\subseteq_{[]}(\_), h_1) \equiv \mathrm{disj}_{[]}(l_3)$

$\quad \subseteq_{\wedge}\mathrm{disj}_{to}\mathrm{disj}(\subseteq_{:}(h_2, h_3), h_1) \equiv \mathrm{disj}_{:}(\subseteq_{\wedge}\mathrm{disj}_{to}\mathrm{disj}(h_2, h_1), \in_{\wedge}\mathrm{disj}_{to}\notin(h_3, h_1))$

C.3.3 Lists of Variables

Var ∈ **Set**
  Var ≡ N
ListVar ∈ **Set**
  ListVar ≡ List(Var)
Var_{dec} ∈ (*x*, *y* ∈ Var) Dec(=(*x*, *y*))
  Var_{dec} ≡ N_{dec}
$Var_{to}A$ ∈ (*a*, *b* ∈ *A*; Dec(=(*x*, *y*))) *A*
  $Var_{to}A$(*a*, *b*, yes(h_1)) ≡ *a*
  $Var_{to}A$(*a*, *b*, no(h_1)) ≡ *b*
$=var_{to}A$ ∈ (*a*, *b* ∈ *A*; *p* ∈ Dec(=(*x*, *x*))) =(*a*, $Var_{to}A$(*a*, *b*, *p*))
  $=var_{to}A$(*a*, *b*, yes(*h*)) ≡ refl(*a*)
  $=var_{to}A$(*a*, *b*, no($\Rightarrow_I$(*f*))) ≡ **case** *f*(refl(*x*)) ∈ ⊥ **of**
                end
$\neq var_{to}A$ ∈ (*a*, *b* ∈ *A*; ¬(=(*x*, *y*)); *p* ∈ Dec(=(*x*, *y*))) =(*b*, $Var_{to}A$(*a*, *b*, *p*))
  $\neq var_{to}A$(*a*, *b*, $\Rightarrow_I$(*f*), yes(h_1)) ≡ **case** *f*(h_1) ∈ ⊥ **of**
                end
  $\neq var_{to}A$(*a*, *b*, *h*, no(h_1)) ≡ refl(*b*)
$\in_{dec}$ ∈ (*x* ∈ Var; *l* ∈ ListVar) Dec($\in_L$(*x*, *l*))
  $\in_{dec}$(*x*, []) ≡ no($\Rightarrow_I$($\in_{[]to}\perp$))
  $\in_{dec}$(*x*, :(l_1, *y*)) ≡ **case** Var_{dec}(*x*, *y*) ∈ Dec(=(*x*, *y*)) **of**
                yes(refl(_)) ⇒ yes($\in_{hd}$(*y*, l_1))
                no(*h*) ⇒ **case** $\in_{dec}$(*x*, l_1) ∈ Dec($\in_L$(*x*, l_1)) **of**
                yes(h_1) ⇒ yes($\in_{tl}$(*y*, l_1, h_1))
                no(h_1) ⇒ no($\Rightarrow_I$($\in_{:to}\perp$(*h*, h_1)))
                end
        end
$\in_{to}$N ∈ (*n*, *m* ∈ N; Dec($\in_L$(*x*, *l*))) N
  $\in_{to}$N(*n*, *m*, yes(h_1)) ≡ *n*
  $\in_{to}$N(*n*, *m*, no(h_1)) ≡ *m*
len ∈ (*l* ∈ ListVar) N
  len([]) ≡ 0
  len(:(l_1, *x*)) ≡ $\in_{to}$N(len(l_1), +(len(l_1), 1), $\in_{dec}$(*x*, l_1))
$\in_{to}$len ∈ (*n*, *m* ∈ N; $\in_L$(*x*, *l*); *p* ∈ Dec($\in_L$(*x*, *l*))) =(*n*, $\in_{to}$N(*n*, *m*, *p*))
  $\in_{to}$len(*n*, *m*, *h*, yes(h_1)) ≡ refl(*n*)
  $\in_{to}$len(*n*, *m*, *h*, no($\Rightarrow_I$(*f*))) ≡ **case** *f*(*h*) ∈ ⊥ **of**
                end
$\neg\in_{to}$len ∈ (*n*, *m* ∈ N; ¬($\in_L$(*x*, *l*)); *p* ∈ Dec($\in_L$(*x*, *l*))) =(*m*, $\in_{to}$N(*n*, *m*, *p*))
  $\neg\in_{to}$len(*n*, *m*, $\Rightarrow_I$(*f*), yes(h_1)) ≡ **case** *f*(h_1) ∈ ⊥ **of**
                end
  $\neg\in_{to}$len(*n*, *m*, *h*, no(h_1)) ≡ refl(*m*)
$=_{len}$ ∈ ($\in_L$(*x*, *l*)) =(len(*l*), len(:(*l*, *x*)))
  $=_{len}$(*h*) ≡ $\in_{to}$len(len(*l*), +(len(*l*), 1), *h*, $\in_{dec}$(*x*, *l*))
$=_{len+1}$ ∈ (¬($\in_L$(*x*, *l*))) =(+(len(*l*), 1), len(:(*l*, *x*)))
  $=_{len+1}$(*h*) ≡ ¬$\in_{to}$len(len(*l*), +(len(*l*), 1), *h*, $\in_{dec}$(*x*, *l*))

```
0<len ∈ (l ∈ ListVar; ∈L(x, l)) <(0, len(l))
    0<len([], h) ≡ case h ∈ ∈L(x, []) of
                   end
    0<len(:(l1, y), h) ≡ case ∈dec(y, l1) ∈ Dec(∈L(y, l1)) of
                             yes(h1) ⇒ =subst1(=len(h1), 0<len(l1, h1))
                             no(h1) ⇒ =subst1(=len+1(h1), <0(len(l1)))
                         end
/ ∈ (l ∈ ListVar; x ∈ Var) ListVar
    /([], x) ≡ []
    /(:(l1, y), x) ≡ Var_toA(/(l1, x), :(/(l1, x), y), Var_dec(y, x))
/=var ∈ (x ∈ Var; l ∈ ListVar) =(/(l, x), /(:(l, x), x))
    /=var(x, l) ≡ =var_toA(/(l, x), :(/(l, x), x), Var_dec(x, x))
/≠var ∈ (l ∈ ListVar; ¬(=(x, y))) =(:(/(l, y), x), /(:(l, x), y))
    /≠var(l, h) ≡ ≠var_toA(/(l, y), :(/(l, y), x), h, Var_dec(x, y))
∈/ ∈ (¬(=(x, y)); ∈L(x, l)) ∈L(x, /(l, y))
    ∈/(h, ∈hd(–, l1)) ≡ =subst1(/≠var(l1, h), ∈hd(x, /(l1, y)))
    ∈/(h, ∈tl(z, l1, h2)) ≡ case Var_dec(z, y) ∈ Dec(=(z, y)) of
                               yes(refl(–)) ⇒ =subst1(/=var(y, l1), ∈/(h, h2))
                               no(h1) ⇒ =subst1(/≠var(l1, h1), ∈tl(z, /(l1, y), ∈/(h, h2)))
                           end
⊆/ ∈ (∉L(x, l1); ⊆(l1, l2)) ⊆(l1, /(l2, x))
    ⊆/(h, ⊆[](–)) ≡ ⊆[](/(l2, x))
    ⊆/(∉:(h1, ⇒I(f)), ⊆:(h2, h3)) ≡ ⊆:(⊆/(h1, h2), ∈/(⇒I([h]f(=symm(h))), h3))
∉/ ∈ (y ∈ Var; ∉L(x, l)) ∉L(x, /(l, y))
    ∉/(y, ∉[](–)) ≡ ∉[](x)
    ∉/(y, ∉:(h1, h2)) ≡ case Var_dec(z, y) ∈ Dec(=(z, y)) of
                           yes(refl(–)) ⇒ =subst1(/=var(y, l1), ∉/(y, h1))
                           no(h) ⇒ =subst1(/≠var(l1, h), ∉:(∉/(y, h1), h2))
                       end
∉to=len ∈ (∉L(x, l)) =(len(l), len(/(l, x)))
    ∉to=len(∉[](–)) ≡ refl(len([]))
    ∉to=len(∉:(h1, ⇒I(f))) ≡
        case ∈dec(y, l1) ∈ Dec(∈L(y, l1)) of
            yes(h) ⇒
                =subst2(=symm(=len(h)),
                        =trans(=len(∈/(⇒I([h2]f(=symm(h2))), h)),
                               =cong1(len, /≠var(l1, ⇒I([h2]f(=symm(h2))))))),
                        ∉to=len(h1))
            no(h) ⇒
                =subst2(=symm(=len+1(h)),
                        =trans(=len+1(∉to¬∈(∉/(x, ¬∈to∉(l1, h))))),
                               =cong1(len, /≠var(l1, ⇒I([h2]f(=symm(h2))))))),
                        scong(∉to=len(h1)))
        end
```

```
∈to=len+1 ∈ (l ∈ ListVar; ∈L(x, l)) =(len(l), +(len(/(l, x)), 1))
    ∈to=len+1([], h) ≡ case h ∈ ∈L(x, []) of
                          end
    ∈to=len+1(:(l1, y), h) ≡
        case Vardec(x, y) ∈ Dec(=(x, y)) of
            yes(refl(_)) ⇒
                case ∈dec(y, l1) ∈ Dec(∈L(y, l1)) of
                    yes(h2) ⇒
                        =subst2(=symm(=len(h2)),
                                =cong1([n]+(n, 1), =cong1(len, /=var(y, l1))),
                                ∈to=len+1(l1, h2))
                    no(h2) ⇒
                        =subst2(=symm(=len+1(h2)),
                                =cong1([n]+(n, 1), =cong1(len, /=var(y, l1))),
                                =cong1([n]+(n, 1), ∉to=len(¬∈to∉(l1, h2))))
                end
            no(⇒I(f)) ⇒
                case ∈dec(y, l1) ∈ Dec(∈L(y, l1)) of
                    yes(h1) ⇒
                        =subst2(=symm(=len(h1)),
                                =cong1([n]+(n, 1),
                                       =subst1(/≠var(l1, ⇒I([h2]f(=symm(h2)))),
                                               =len(∈/(⇒I([h2]f(=symm(h2))), h1)))),
                                ∈to=len+1(l1, ∈:red(⇒I(f), h)))
                    no(h1) ⇒
                        =subst2(=symm(=len+1(h1)),
                                =cong1([n]+(n, 1),
                                       =subst1(/≠var(l1, ⇒I([h2]f(=symm(h2)))),
                                               =len+1(∉to¬∈(∉/(x, ¬∈to∉(l1, h1)))))),
                                =cong1([n]+(n, 1), ∈to=len+1(l1, ∈:red(⇒I(f), h))))
                end
        end
⊂to< ∈ (l1 ∈ ListVar; ¬(∈L(x, l1)); ∈L(x, l2); ⊆(l1, l2)) <(len(l1), len(l2))
    ⊂to<([], h, h1, h2) ≡ 0<len(l2, h1)
    ⊂to<(:(l3, y), h, h1, ⊆:(h3, h4)) ≡
        case ¬∈:to∧(h) ∈ ∧(¬(=(x, y)), ¬(∈L(x, l3))) of
            ∧I(h2, h5) ⇒
                case ∈dec(y, l3) ∈ Dec(∈L(y, l3)) of
                    yes(h6) ⇒ =subst1(=len(h6), ⊂to<(l3, h5, h1, h3))
                    no(h6) ⇒
                        =subst2(=symm(=len+1(h6)),
                                =symm(∈to=len+1(l2, h4)),
                                <s(⊂to<(l3, h5, ∈/(h2, h1), ⊆/(¬∈to∉(l3, h6), h3))))
                end
        end
⊆to≤ ∈ (⊆(l1, l2)) ≤(len(l1), len(l2))
    ⊆to≤(⊆[](_)) ≡ 0≤(len(l2))
    ⊆to≤(⊆:(h1, h2)) ≡ case ∈dec(x, l3) ∈ Dec(∈L(x, l3)) of
                           yes(h) ⇒ =subst1(=len(h), ⊆to≤(h1))
                           no(h) ⇒ =subst1(=len+1(h), <to≤sR(⊂to<(l3, h, h2, h1)))
                       end
```

C.3.4 Natural Numbers

```
N ∈ Set
    0 ∈ N
    s ∈ (n ∈ N) N
+ ∈ (n, m ∈ N) N
    +(n, 0) ≡ n
    +(n, s(m1)) ≡ s(+(n, m1))
< ∈ (n, m ∈ N) Set
    <0 ∈ (m ∈ N) <(0, s(m))
    <s ∈ (<(n, m)) <(s(n), s(m))
≤ ∈ (n, m ∈ N) Set
    ≤ ≡ [n, m]∨(<(n, m), =(n, m))
1 ∈ N
    1 ≡ s(0)
scong ∈ (=(n, m)) =(s(n), s(m))
    scong(h) ≡ =cong1(s, h)
sinj ∈ (=(s(n), s(m))) =(n, m)
    sinj(refl(–)) ≡ refl(m)
0=sto⊥ ∈ (=(0, s(n))) ⊥
    0=sto⊥(h) ≡ case h ∈ =(0, s(n)) of
                end
s=0to⊥ ∈ (=(s(n), 0)) ⊥
    s=0to⊥(h) ≡ case h ∈ =(s(n), 0) of
                end
snsmto⊥ ∈ (¬(=(n, m)); =(s(n), s(m))) ⊥
    snsmto⊥(⇒I(f), h1) ≡ f(sinj(h1))
¬0=s ∈ ¬(=(0, s(n)))
    ¬0=s ≡ ⇒I(0=sto⊥)
¬s=0 ∈ ¬(=(s(n), 0))
    ¬s=0 ≡ ⇒I(s=0to⊥)
¬snsm ∈ (¬(=(n, m))) ¬(=(s(n), s(m)))
    ¬snsm(h) ≡ ⇒I(snsmto⊥(h))
Ndec ∈ (n, m ∈ N) Dec(=(n, m))
    Ndec(0, 0) ≡ yes(refl(0))
    Ndec(0, s(m1)) ≡ no(¬0=s)
    Ndec(s(n1), 0) ≡ no(¬s=0)
    Ndec(s(n1), s(m1)) ≡ case Ndec(n1, m1) ∈ Dec(=(n1, m1)) of
                            yes(h) ⇒ yes(scong(h))
                            no(h) ⇒ no(¬snsm(h))
                         end
0+ ∈ (n ∈ N) =(+(0, n), n)
    0+(0) ≡ refl(0)
    0+(s(n1)) ≡ =cong1(s, 0+(n1))
s+ ∈ (n, m ∈ N) =(+(s(n), m), s(+(n, m)))
    s+(n, 0) ≡ refl(s(n))
    s+(n, s(m1)) ≡ =cong1(s, s+(n, m1))
```

$+_{comm} \in (n, m \in N) =(+(n, m), +(m, n))$
$+_{comm}(0, m) \equiv 0+(m)$
$+_{comm}(s(n_1), m) \equiv =_{trans}(s+(n_1, m), =_{cong1}(s, +_{comm}(n_1, m)))$
$+_{assoc} \in (n, m, p \in N) =(+(+(n, m), p), +(n, +(m, p)))$
$+_{assoc}(n, m, 0) \equiv refl(+(n, m))$
$+_{assoc}(n, m, s(p_1)) \equiv =_{cong1}(s, +_{assoc}(n, m, p_1))$
$+_{assoc\_comm} \in (n, m, p, q \in N) =(+(+(n, m), +(p, q)), +(+(n, p), +(m, q)))$
$+_{assoc\_comm}(n, m, p, q) \equiv$
$=_{subst2}(+_{assoc}(n, m, +(p, q)),$
$=_{symm}(+_{assoc}(n, p, +(m, q))),$
$=_{cong1}(+(n),$
$=_{subst2}(=_{symm}(+_{assoc}(m, p, q)),$
$+_{assoc}(p, m, q),$
$=_{cong1}([r]+(r, q), +_{comm}(m, p)))))$
$0\leq \in (n \in N) \leq(0, n)$
$0\leq(0) \equiv \vee_R(refl(0))$
$0\leq(s(n)) \equiv \vee_L(<_0(n))$
$\leq_{s\_mon} \in (\leq(n, m)) \leq(s(n), s(m))$
$\leq_{s\_mon}(\vee_L(h_1)) \equiv \vee_L(<_s(h_1))$
$\leq_{s\_mon}(\vee_R(h_1)) \equiv \vee_R(s_{cong}(h_1))$
$<_{to}\leq_{sR} \in (<(n, m)) \leq(s(n), m)$
$<_{to}\leq_{sR}(<_0(m_1)) \equiv \leq_{s\_mon}(0\leq(m_1))$
$<_{to}\leq_{sR}(<_s(h_1)) \equiv$ **case** $<_{to}\leq_{sR}(h_1) \in \leq(s(n_1), m_1)$ **of**
$\vee_L(h) \Rightarrow \vee_L(<_s(h))$
$\vee_R(h) \Rightarrow \vee_R(s_{cong}(h))$
end
$<_{to}\leq_{sL} \in (<(n, s(m))) \leq(n, m)$
$<_{to}\leq_{sL}(<_0(\_)) \equiv 0\leq(m)$
$<_{to}\leq_{sL}(<_s(h_1)) \equiv <_{to}\leq_{sR}(h_1)$
$acc_{N\_aux1} \in (<(n, 0)) Acc(N, <, n)$
$acc_{N\_aux1}(h) \equiv$ **case** $h \in <(n, 0)$ **of**
end
$acc_{N\_aux2} \in (Acc(N, <, m); \leq(n, m)) Acc(N, <, n)$
$acc_{N\_aux2}(acc(\_, p), \vee_L(h_2)) \equiv p(n, h_2)$
$acc_{N\_aux2}(h, \vee_R(refl(\_))) \equiv h$
$allacc_N \in (n \in N) Acc(N, <, n)$
$allacc_N(0) \equiv acc(0, acc_{N\_aux1})$
$allacc_N(s(n_1)) \equiv acc(s(n_1), [m, h]acc_{N\_aux2}(allacc_N(n_1), <_{to}\leq_{sL}(h)))$
$WF_N \in WF(N, <)$
$WF_N \equiv \forall_I(allacc_N)$
$<_{trans} \in (<(n, m); <(m, p)) <(n, p)$
$<_{trans}(h, <_0(p_1)) \equiv$ **case** $h \in <(n, 0)$ **of**
end
$<_{trans}(<_0(\_), <_s(h_2)) \equiv <_0(p_1)$
$<_{trans}(<_s(h_1), <_s(h_2)) \equiv <_s(<_{trans}(h_1, h_2))$
$<_{sR} \in (n \in N) <(n, s(n))$
$<_{sR}(0) \equiv <_0(0)$
$<_{sR}(s(n_1)) \equiv <_s(<_{sR}(n_1))$

$<_{ssR} \in (n \in N) <(n, s(s(n)))$
$<_{ssR}(n) \equiv <_{trans}(<_{sR}(n), <_{sR}(s(n)))$
$\leq_{to}<_{sL} \in (\leq(n, m)) <(n, s(m))$
$\leq_{to}<_{sL}(\vee_L(h)) \equiv <_{trans}(h, <_{sR}(m))$
$\leq_{to}<_{sL}(\vee_R(\text{refl}(\_))) \equiv <_{sR}(m)$
$<\wedge=_{to}\bot \in (<(n, m); =(n, m)) \bot$
$<\wedge=_{to}\bot(<_0(m_1), h_1) \equiv$ **case** $h_1 \in =(0, s(m_1))$ **of**
end
$<\wedge=_{to}\bot(<_s(h_2), h_1) \equiv <\wedge=_{to}\bot(h_2, s_{inj}(h_1))$
$\leq\wedge\leq_{to}= \in (\leq(n, m); \leq(m, n)) =(n, m)$
$\leq\wedge\leq_{to}=(\vee_L(<_0(m_1)), \vee_L(h)) \equiv$ **case** $h \in <(s(m_1), 0)$ **of**
end
$\leq\wedge\leq_{to}=(\vee_L(<_s(h_1)), \vee_L(<_s(h_2))) \equiv s_{cong}(\leq\wedge\leq_{to}=(\vee_L(h_1), \vee_L(h_2)))$
$\leq\wedge\leq_{to}=(\vee_L(h_2), \vee_R(h)) \equiv$ **case** $<\wedge=_{to}\bot(h_2, =_{symm}(h)) \in \bot$ **of**
end
$\leq\wedge\leq_{to}=(\vee_R(h_2), h_1) \equiv h_2$
$<\wedge\leq_{trans} \in (<(n, m); \leq(m, p)) <(n, p)$
$<\wedge\leq_{trans}(h, \vee_L(h_1)) \equiv <_{trans}(h, h_1)$
$<\wedge\leq_{trans}(h, \vee_R(\text{refl}(\_))) \equiv h$
$\leq\wedge<_{trans} \in (\leq(n, m); <(m, p)) <(n, p)$
$\leq\wedge<_{trans}(\vee_L(h), h_1) \equiv <_{trans}(h, h_1)$
$\leq\wedge<_{trans}(\vee_R(\text{refl}(\_)), h_1) \equiv h_1$
$<_{monR} \in (p \in N; <(n, m)) <(+(n, p), +(m, p))$
$<_{monR}(0, h) \equiv h$
$<_{monR}(s(p_1), h) \equiv <_s(<_{monR}(p_1, h))$
$\leq_{to}\leq_{+R} \in (p \in N; \leq(n, m)) \leq(n, +(m, p))$
$\leq_{to}\leq_{+R}(0, h) \equiv h$
$\leq_{to}\leq_{+R}(s(p_1), h) \equiv \vee_L(\leq\wedge<_{trans}(\leq_{to}\leq_{+R}(p_1, h), <_{sR}(+(m, p_1))))$
$\leq_{+L} \in (n, m \in N) \leq(n, +(m, n))$
$\leq_{+L}(n, 0) \equiv \vee_R(=_{symm}(0+(n)))$
$\leq_{+L}(n, s(m_1)) \equiv =_{subst1}(=_{symm}(s+(m_1, n)), \vee_L(\leq\wedge<_{trans}(\leq_{+L}(n, m_1), <_{sR}(+(m_1, n)))))$

C.3.5 Triplets of Natural Numbers

N3 $\in$ **Set**
N3 $\equiv$ Triple(N, N, N)
$<_{N3} \in (n, m \in N3)$ **Set**
$<_{fst} \in (n_2, m_2, n_3, m_3 \in N; <(n_1, m_1)) <_{N3}(.(n_1, n_2, n_3), .(m_1, m_2, m_3))$
$<_{snd} \in (n_1, n_3, m_3 \in N; <(n_2, m_2)) <_{N3}(.(n_1, n_2, n_3), .(n_1, m_2, m_3))$
$<_{rd} \in (n_1, n_2 \in N; <(n_3, m_3)) <_{N3}(.(n_1, n_2, n_3), .(n_1, n_2, m_3))$
$<_{snd'} \in (n_3, m_3 \in N; =(n_1, m_1); <(n_2, m_2)) <_{N3}(.(n_1, n_2, n_3), .(m_1, m_2, m_3))$
$<_{snd'}(n_3, m_3, \text{refl}(\_), h_1) \equiv <_{snd}(m_1, n_3, m_3, h_1)$
$<_{rd'} \in (=(n_1, m_1); =(n_2, m_2); <(n_3, m_3)) <_{N3}(.(n_1, n_2, n_3), .(m_1, m_2, m_3))$
$<_{rd'}(\text{refl}(\_), \text{refl}(\_), h_2) \equiv <_{rd}(m_1, m_2, h_2)$
$<_{fst\_rd'} \in (\leq(n_1, m_1); =(n_2, m_2); <(n_3, m_3)) <_{N3}(.(n_1, n_2, n_3), .(m_1, m_2, m_3))$
$<_{fst\_rd'}(\vee_L(h_3), h_1, h_2) \equiv <_{fst}(n_2, m_2, n_3, m_3, h_3)$
$<_{fst\_rd'}(\vee_R(h_3), h_1, h_2) \equiv <_{rd'}(h_3, h_1, h_2)$

$acc_{N3\_aux} \in (m_1, m_2, m_3 \in N;$
$h_1 \in (x \in N; <(x, m_1)) \forall(N, [p]\forall(N, [q]Acc(N3, <_{N3}, .(x, p, q))));$
$h_2 \in (y \in N; <(y, m_2)) \forall(N, [q]Acc(N3, <_{N3}, .(m_1, y, q)));$
$h_3 \in (z \in N; <(z, m_3)) Acc(N3, <_{N3}, .(m_1, m_2, z));$
$n \in N3;$
$<_{N3}(n, .(m_1, m_2, m_3))$
$) Acc(N3, <_{N3}, n)$
$acc_{N3\_aux}(m_1, m_2, m_3, h_1, h_2, h_3, \_, <_{fst}(n_2, \_, n_3, \_, h_4)) \equiv \forall_E(\forall_E(h_1(n_1, h_4), n_2), n_3)$
$acc_{N3\_aux}(m_1, m_2, m_3, h_1, h_2, h_3, \_, <_{snd}(\_, n_3, \_, h_4)) \equiv \forall_E(h_2(n_2, h_4), n_3)$
$acc_{N3\_aux}(m_1, m_2, m_3, h_1, h_2, h_3, \_, <_{rd}(\_, \_, h_4)) \equiv h_3(n_3, h_4)$

$acc_{N3\_xyz} \in (x, y, z \in N;$
$h_1 \in (x' \in N; <(x', x)) \forall(N, [p]\forall(N, [q]Acc(N3, <_{N3}, .(x', p, q))));$
$h_2 \in (y' \in N; <(y', y)) \forall(N, [q]Acc(N3, <_{N3}, .(x, y', q)));$
$h_3 \in (z' \in N; <(z', z)) Acc(N3, <_{N3}, .(x, y, z'))$
$) Acc(N3, <_{N3}, .(x, y, z))$
$acc_{N3\_xyz}(x, y, z, h_1, h_2, h_3) \equiv acc(.(x, y, z), acc_{N3\_aux}(x, y, z, h_1, h_2, h_3))$

$acc_{N3\_xy} \in (x, y \in N;$
$h_1 \in (x' \in N; <(x', x)) \forall(N, [p]\forall(N, [q]Acc(N3, <_{N3}, .(x', p, q))));$
$h_2 \in (y' \in N; <(y', y)) \forall(N, [q]Acc(N3, <_{N3}, .(x, y', q)))$
$) \forall(N, [q]Acc(N3, <_{N3}, .(x, y, q)))$
$acc_{N3\_xy}(x, y, h_1, h_2) \equiv$
$\forall_I([q]wfrec(q, allacc_N(q), [z, h, h_3]acc_{N3\_xyz}(x, y, z, h_1, h_2, h_3)))$

$acc_{N3\_x} \in (x \in N;$
$h_1 \in (x' \in N; <(x', x)) \forall(N, [p]\forall(N, [q]Acc(N3, <_{N3}, .(x', p, q))))$
$) \forall(N, [p]\forall(N, [q]Acc(N3, <_{N3}, .(x, p, q))))$
$acc_{N3\_x}(x, h_1) \equiv \forall_I([p]wfrec(p, allacc_N(p), [y, h, h_2]acc_{N3\_xy}(x, y, h_1, h_2)))$

$acc_{N3} \in (n_1 \in N) \forall(N, [p]\forall(N, [q]Acc(N3, <_{N3}, .(n_1, p, q))))$
$acc_{N3}(n_1) \equiv wfrec(n_1, allacc_N(n_1), [x, h, h_1]acc_{N3\_x}(x, h_1))$

$allacc_{N3} \in (n \in N3) Acc(N3, <_{N3}, n)$
$allacc_{N3}(.(n_1, n_2, n_3)) \equiv \forall_E(\forall_E(acc_{N3}(n_1), n_2), n_3)$

$WF_{N3} \in WF(N3, <_{N3})$
$WF_{N3} \equiv \forall_I(allacc_{N3})$

C.3.6 Tuples

$Pair \in (A, B \in \mathbf{Set})\ \mathbf{Set}$
$. \in (a \in A;\ b \in B)\ Pair(A, B)$

$Triple \in (A, B, C \in \mathbf{Set})\ \mathbf{Set}$
$. \in (a \in A;\ b \in B;\ c \in C)\ Triple(A, B, C)$

C.3.7 Vector

$Vector \in (n \in N;\ A \in \mathbf{Set})\ \mathbf{Set}$
$[]_v \in Vector(0, A)$
$:_v \in (v \in Vector(n, A);\ a \in A)\ Vector(s(n), A)$

$=_{vector to =_{head}} \in (=(:_v(v_1, a), :_v(v_2, b))) =(a, b)$
$=_{vector to =_{head}}(refl(\_)) \equiv refl(b)$

$=_{vector to =_{tail}} \in (=(:_v(v_1, a), :_v(v_2, b))) =(v_1, v_2)$
$=_{vector to =_{tail}}(refl(\_)) \equiv refl(v_2)$

C.4 Terms, List of Pairs of Terms and Substitutions

C.4.1 Terms

Fun ∈ **Set**
  Fun ≡ N
Fun_{dec} ∈ (*f*, *g* ∈ Fun) Dec(=(*f*, *g*))
  Fun_{dec} ≡ N_{dec}
Term ∈ **Set**
  var ∈ (*x* ∈ Var) Term
  fun ∈ (*f* ∈ Fun; *lt* ∈ Vector(*n*, Term)) Term
VTerm ∈ (*n* ∈ N) **Set**
  VTerm(*n*) ≡ Vector(*n*, Term)
$=_{T to} \bot$ ∈ (=(var(*x*), fun(*f*, *lt*))) ⊥
  $=_{T to} \bot(h)$ ≡ **case** *h* ∈ =(var(*x*), fun(*f*, *lt*)) **of**
    end
$\neq_T$ ∈ (*f* ∈ Fun; *x* ∈ Var; *lt* ∈ VTerm(*n*)) ¬(=(var(*x*), fun(*f*, *lt*)))
  $\neq_T(f, x, lt)$ ≡ $\Rightarrow_I(=_{T to} \bot)$
$=_{fun to =_f}$ ∈ (=(fun(*f*, lt_1), fun(*g*, lt_2))) =(*f*, *g*)
  $=_{fun to =_f}$(refl(_)) ≡ refl(*g*)
$=_{fun to =_{arity}}$ ∈ (=(fun(*f*, lt_1), fun(*g*, lt_2))) =(n_1, n_2)
  $=_{fun to =_{arity}}$(refl(_)) ≡ refl(n_2)
$=_{fun to =_{vector}}$ ∈ (=(fun(*f*, lt_1), fun(*g*, lt_2))) =(lt_1, lt_2)
  $=_{fun to =_{vector}}$(refl(_)) ≡ refl(lt_2)
$=_{T to =_{Var}}$ ∈ (=(var(x_1), var(x_2))) =(x_1, x_2)
  $=_{T to =_{Var}}$(refl(_)) ≡ refl(x_2)
$vars_T$ ∈ (*t* ∈ Term) ListVar
  $vars_T$(var(*x*)) ≡ :([], *x*)
  $vars_T$(fun(*f*, *lt*)) ≡ $vars_{VT}$(*lt*)
$vars_{VT}$ ∈ (*lt* ∈ VTerm(*n*)) ListVar
  $vars_{VT}([]_v)$ ≡ []
  $vars_{VT}(:_v(lt', t))$ ≡ ++($vars_{VT}$(*lt'*), $vars_T$(*t*))
$\#funs_T$ ∈ (*t* ∈ Term) N
  $\#funs_T$(var(*x*)) ≡ 0
  $\#funs_T$(fun(*f*, *lt*)) ≡ +($\#funs_{VT}$(*lt*), 1)
$\#funs_{VT}$ ∈ (*lt* ∈ VTerm(*n*)) N
  $\#funs_{VT}([]_v)$ ≡ 0
  $\#funs_{VT}(:_v(lt', t))$ ≡ +($\#funs_{VT}$(*lt'*), $\#funs_T$(*t*))
$:=_T$ ∈ (*x* ∈ Var; *t*, t_1 ∈ Term) Term
  $:=_T(x, t, \text{var}(x_1))$ ≡ $\text{Var}_{to}\text{A}(t, \text{var}(x_1), \text{Var}_{dec}(x, x_1))$
  $:=_T(x, t, \text{fun}(f, lt))$ ≡ fun(*f*, $:=_{VT}(x, t, lt)$)
$:=_{VT}$ ∈ (*x* ∈ Var; *t* ∈ Term; *lt* ∈ VTerm(*n*)) VTerm(*n*)
  $:=_{VT}(x, t, []_v)$ ≡ $[]_v$
  $:=_{VT}(x, t, :_v(lt_1, t_1))$ ≡ $:_v(:=_{VT}(x, t, lt_1), :=_T(x, t, t_1))$
$:=_{=var}$ ∈ (*x* ∈ Var; *t* ∈ Term) =(*t*, $:=_T(x, t, \text{var}(x))$)
  $:=_{=var}(x, t)$ ≡ $=\text{var}_{to}\text{A}(t, \text{var}(x), \text{Var}_{dec}(x, x))$
$:=_{\neq var}$ ∈ (*t* ∈ Term; ¬(=(*x*, *y*))) =(var(*y*), $:=_T(x, t, \text{var}(y))$)
  $:=_{\neq var}(t, h)$ ≡ $\neq\text{var}_{to}\text{A}(t, \text{var}(y), h, \text{Var}_{dec}(x, y))$

C.4.2 Properties of Terms

$=_{:=T} \in (t, t_1 \in \text{Term};\ \notin_L(x, \text{vars}_T(t_1))) = (t_1, :=_T(x, t, t_1))$

$=_{:=T}(t, \text{var}(x_1), h) \equiv :=_{\neq\text{var}}(t, \notin_{\text{to}\neq}(h))$

$=_{:=T}(t, \text{fun}(f, lt), h) \equiv =_{\text{cong1}}(\text{fun}(f), =_{:=VT}(t, lt, h))$

$=_{:=VT} \in (t \in \text{Term};\ lt \in \text{VTerm}(n);\ \notin_L(x, \text{vars}_{VT}(lt))) = (lt, :=_{VT}(x, t, lt))$

$=_{:=VT}(t, []_v, h) \equiv \text{refl}([]_v)$

$=_{:=VT}(t, :_v(lt', t'), h) \equiv$
$\quad =_{\text{cong2}}(:_v, =_{:=VT}(t, lt', \notin{+}{+}_{\text{redR}}(\text{vars}_T(t'), h)), =_{:=T}(t, t', \notin{+}{+}_{\text{redL}}(\text{vars}_T(t'), h)))$

$\notin :=_{I=\text{var}} \in (t' \in \text{Term};\ \notin_L(x, \text{vars}_T(t)))\ \notin_L(x, \text{vars}_T(:=_T(x, t, t')))$

$\notin :=_{I=\text{var}}(\text{var}(x_1), h) \equiv$ **case** $\text{Var}_{\text{dec}}(x, x_1) \in \text{Dec}(=(x, x_1))$ **of**
$\quad \text{yes}(\text{refl}(\_)) \Rightarrow =_{\text{subst1}}(=_{\text{cong1}}(\text{vars}_T, :=_{=\text{var}}(x_1, t)), h)$
$\quad \text{no}(h_1) \Rightarrow =_{\text{subst1}}(=_{\text{cong1}}(\text{vars}_T, :=_{\neq\text{var}}(t, h_1)), \notin_{:}(\notin_{[]}(x), h_1))$
end

$\notin :=_{I=\text{var}}(\text{fun}(f, lt), h) \equiv \notin :=_{VT=\text{var}}(lt, h)$

$\notin :=_{VT=\text{var}} \in (lt \in \text{VTerm}(n);\ \notin_L(x, \text{vars}_T(t)))\ \notin_L(x, \text{vars}_{VT}(:=_{VT}(x, t, lt)))$

$\notin :=_{VT=\text{var}}([]_v, h) \equiv \notin_{[]}(x)$

$\notin :=_{VT=\text{var}}(:_v(lt', t'), h) \equiv \notin{+}{+}(\notin :=_{VT=\text{var}}(lt', h), \notin :=_{I=\text{var}}(t', h))$

$\notin :=_{I\neq\text{var}} \in (y \in \text{Var};\ t' \in \text{Term};\ \notin_L(x, \text{vars}_T(t));\ \notin_L(x, \text{vars}_T(t')))\ \notin_L(x, \text{vars}_T(:=_T(y, t, t')))$

$\notin :=_{I\neq\text{var}}(y, \text{var}(x_1), h, h_1) \equiv$ **case** $\text{Var}_{\text{dec}}(y, x_1) \in \text{Dec}(=(y, x_1))$ **of**
$\quad \text{yes}(\text{refl}(\_)) \Rightarrow =_{\text{subst1}}(=_{\text{cong1}}(\text{vars}_T, :=_{=\text{var}}(x_1, t)), h)$
$\quad \text{no}(h_2) \Rightarrow =_{\text{subst1}}(=_{\text{cong1}}(\text{vars}_T, :=_{\neq\text{var}}(t, h_2)), h_1)$
end

$\notin :=_{I\neq\text{var}}(y, \text{fun}(f, lt), h, h_1) \equiv \notin :=_{VT\neq\text{var}}(y, lt, h, h_1)$

$\notin :=_{VT\neq\text{var}} \in (y \in \text{Var};$
$\quad lt \in \text{VTerm}(n);$
$\quad \notin_L(x, \text{vars}_T(t));$
$\quad \notin_L(x, \text{vars}_{VT}(lt))$
$\quad)\ \notin_L(x, \text{vars}_{VT}(:=_{VT}(y, t, lt)))$

$\notin :=_{VT\neq\text{var}}(y, []_v, h, h_1) \equiv \notin_{[]}(x)$

$\notin :=_{VT\neq\text{var}}(y, :_v(lt', t'), h, h_1) \equiv$
$\quad \notin{+}{+}(\notin :=_{VT\neq\text{var}}(y, lt', h, \notin{+}{+}_{\text{redR}}(\text{vars}_T(t'), h_1)), \notin :=_{I\neq\text{var}}(y, t', h, \notin{+}{+}_{\text{redL}}(\text{vars}_T(t'), h_1)))$

$\subseteq\text{vars}_{:=T} \in (x \in \text{Var};\ t' \in \text{Term};\ \subseteq(\text{vars}_T(t), l);\ \subseteq(\text{vars}_T(t'), l))\ \subseteq(\text{vars}_T(:=_T(x, t, t')), l)$

$\subseteq\text{vars}_{:=T}(x, \text{var}(x_1), h, h_1) \equiv$ **case** $\text{Var}_{\text{dec}}(x, x_1) \in \text{Dec}(=(x, x_1))$ **of**
$\quad \text{yes}(\text{refl}(\_)) \Rightarrow =_{\text{subst1}}(:=_{=\text{var}}(x_1, t), h)$
$\quad \text{no}(h_2) \Rightarrow =_{\text{subst1}}(:=_{\neq\text{var}}(t, h_2), h_1)$
end

$\subseteq\text{vars}_{:=T}(x, \text{fun}(f, lt), h, h_1) \equiv \subseteq\text{vars}_{:=VT}(x, lt, h, h_1)$

$\subseteq\text{vars}_{:=VT} \in (x \in \text{Var};\ lt \in \text{VTerm}(n);\ \subseteq(\text{vars}_T(t), l);\ \subseteq(\text{vars}_{VT}(lt), l))\ \subseteq(\text{vars}_{VT}(:=_{VT}(x, t, lt)), l)$

$\subseteq\text{vars}_{:=VT}(x, []_v, h, h_1) \equiv \subseteq_{[]}(l)$

$\subseteq\text{vars}_{:=VT}(x, :_v(lt_1, t_1), h, h_1) \equiv$
$\quad \subseteq{+}{+}_L(\subseteq\text{vars}_{:=VT}(x, lt_1, h, \subseteq{+}{+}_{\text{redR}}(\text{vars}_T(t_1), h_1)), \subseteq\text{vars}_{:=T}(x, t_1, h, \subseteq{+}{+}_{\text{redL}}(h_1)))$

C.4.3 Lists of Pairs of Terms

PairT ∈ **Set**
  PairT ≡ Pair(Term, Term)
ListPT ∈ **Set**
  ListPT ≡ List(PairT)
$vars_{LPT}$ ∈ (*lp* ∈ ListPT) ListVar
  $vars_{LPT}([]) \equiv []$
  $vars_{LPT}(:(lp', .(t_1, t_2))) \equiv ++(vars_{LPT}(lp'), ++(vars_T(t_1), vars_T(t_2)))$
$\#vars_{LPT}$ ∈ (*lp* ∈ ListPT) N
  $\#vars_{LPT}(lp) \equiv len(vars_{LPT}(lp))$
$\#funs_{LPT}$ ∈ (*lp* ∈ ListPT) N
  $\#funs_{LPT}([]) \equiv 0$
  $\#funs_{LPT}(:(lp', .(t_1, t_2))) \equiv +(\#funs_{LPT}(lp'), +(\#funs_T(t_1), \#funs_T(t_2)))$
$\#eqs_{LPT}$ ∈ (*lp* ∈ ListPT) N
  $\#eqs_{LPT}([]) \equiv 0$
  $\#eqs_{LPT}(:(lp', .(var(x), var(x_1)))) \equiv Var_{to}A(+(\#eqs_{LPT}(lp'), 1), \#eqs_{LPT}(lp'), Var_{dec}(x, x_1))$
  $\#eqs_{LPT}(:(lp', .(var(x), fun(f, lt)))) \equiv \#eqs_{LPT}(lp')$
  $\#eqs_{LPT}(:(lp', .(fun(f, lt), var(x)))) \equiv +(\#eqs_{LPT}(lp'), 1)$
  $\#eqs_{LPT}(:(lp', .(fun(f, lt), fun(f_1, lt_1)))) \equiv \#eqs_{LPT}(lp')$
$LPT_{to}N3$ ∈ (*lp* ∈ ListPT) N3
  $LPT_{to}N3(lp) \equiv .(\#vars_{LPT}(lp), \#funs_{LPT}(lp), \#eqs_{LPT}(lp))$
$:=_{LPT}$ ∈ (*x* ∈ Var; *t* ∈ Term; *lp* ∈ ListPT) ListPT
  $:=_{LPT}(x, t, []) \equiv []$
  $:=_{LPT}(x, t, :(lp', .(t_1, t_2))) \equiv :(:=_{LPT}(x, t, lp'), .(:=_T(x, t, t_1), :=_T(x, t, t_2)))$
zip ∈ (lt_1, lt_2 ∈ VTerm(*n*)) ListPT
  $zip([]_v, []_v) \equiv []$
  $zip(:_v(lt'_1, t_1), :_v(lt'_2, t_2)) \equiv :(zip(lt'_1, lt'_2), .(t_1, t_2))$
$unifies_{LPT}$ ∈ (*sb* ∈ Subst; *lp* ∈ ListPT) **Set**
  $unifies_{LPT\_[]}$ ∈ (*sb* ∈ Subst) $unifies_{LPT}(sb, [])$
  $unifies_{LPT\_:}$ ∈ ($unifies_{LPT}(sb, lp')$;
    $=(appP_T(sb, t_1), appP_T(sb, t_2))$
) $unifies_{LPT}(sb, :(lp', .(t_1, t_2)))$
$\equiv_{LpSbLpSb}$ ∈ (*lp* ∈ ListPT; *sb*, *sb'* ∈ Subst) **Set**
  $\equiv_{LpSbLpSb} \equiv$
    $[lp, sb, sb']\forall(Subst, [sb_1]\Leftrightarrow(\wedge(unifies_{LPT}(sb_1, lp), unifies_S(sb_1, sb)), unifies_S(sb_1, sb')))$
$\equiv_{LpSb}$ ∈ (*lp* ∈ ListPT; *sb* ∈ Subst) **Set**
  $\equiv_{LpSb} \equiv [lp, sb]\equiv_{LpSbLpSb}(lp, [], sb)$
mgu ∈ (*sb* ∈ Subst; *lp* ∈ ListPT) **Set**
  $mgu \equiv [sb, lp]\wedge(unifies_{LPT}(sb, lp), \forall(Subst, [sb']\Rightarrow(unifies_{LPT}(sb', lp), \leq_{Sb}(sb, sb'))))$

C.4.4 Properties of Lists of Pairs of Terms

$\in_{var\_term} \in (x \in Var;\ t \in Term;\ lp \in ListPT) \in_L(x, vars_{LPT}(:(lp, .(var(x), t))))$

$\in_{var\_term}(x, t, lp) \equiv \in{+}{+}_{monL}(vars_{LPT}(lp), \in{+}{+}_{monR}(vars_T(t), \in_{hd}(x, [])))$

$\notin :=_{LPT=var} \in (lp \in ListPT;\ \notin_L(x, vars_T(t))) \notin_L(x, vars_{LPT}(:=_{LPT}(x, t, lp)))$

$\notin :=_{LPT=var}([], h) \equiv \notin_{[]}(x)$

$\notin :=_{LPT=var}(:(lp_1, .(t_1, t_2)), h) \equiv \notin{+}{+}(\notin :=_{LPT=var}(lp_1, h), \notin{+}{+}(\notin :=_{T=var}(t_1, h), \notin :=_{T=var}(t_2, h)))$

$\subseteq vars_{var\_var} \in (x \in Var;\ lp \in ListPT) \subseteq(vars_{LPT}(lp), vars_{LPT}(:(lp, .(var(x), var(x)))))$

$\subseteq vars_{var\_var}(x, lp) \equiv \subseteq{+}{+}_{monR}(+{+}(vars_T(var(x)), vars_T(var(x))), \subseteq_{refl}(vars_{LPT}(lp)))$

$\subseteq vars_{:=_{LPT}} \in (x \in Var;$
$lp \in ListPT;$
$\subseteq(vars_T(t), l);$
$\subseteq(vars_{LPT}(lp), l)$
$) \subseteq(vars_{LPT}(:=_{LPT}(x, t, lp)), l)$

$\subseteq vars_{:=_{LPT}}(x, [], h, h_1) \equiv \subseteq_{[]}(l)$

$\subseteq vars_{:=_{LPT}}(x, :(lp_1, .(t_1, t_2)), h, h_1) \equiv$
$\subseteq{+}{+}_L(\subseteq vars_{:=_{LPT}}(x, lp_1, h, \subseteq{+}{+}_{redR}(+{+}(vars_T(t_1), vars_T(t_2)), h_1)),$
$\subseteq{+}{+}_L(\subseteq vars_{:=_T}(x, t_1, h, \subseteq{+}{+}_{redR}(vars_T(t_2), \subseteq{+}{+}_{redL}(h_1))),$
$\subseteq vars_{:=_T}(x, t_2, h, \subseteq{+}{+}_{redL}(\subseteq{+}{+}_{redL}(h_1)))))$

$\subseteq vars_{var\_term} \in (x \in Var;$
$t \in Term;$
$lp \in ListPT$
$) \subseteq(vars_{LPT}(:=_{LPT}(x, t, lp)), vars_{LPT}(:(lp, .(var(x), t))))$

$\subseteq vars_{var\_term}(x, t, lp) \equiv$
$\subseteq vars_{:=_{LPT}}(x,$
$lp,$
$\subseteq{+}{+}_{monL}(vars_{LPT}(lp), \subseteq{+}{+}_{monL}(vars_T(var(x)), \subseteq_{refl}(vars_T(t)))),$
$\subseteq{+}{+}_{monR}(+{+}(vars_T(var(x)), vars_T(t)), \subseteq_{refl}(vars_{LPT}(lp))))$

$\subseteq vars_{comm} \in (t_1, t_2 \in Term;\ lp \in ListPT) \subseteq(vars_{LPT}(:(lp, .(t_1, t_2)))), vars_{LPT}(:(lp, .(t_2, t_1))))$

$\subseteq vars_{comm}(t_1, t_2, lp) \equiv \subseteq{+}{+}_{mon2L}(vars_{LPT}(lp), \subseteq_{comm}(vars_T(t_1), vars_T(t_2)))$

$\subseteq vars_{var\_fun} \in (f \in Fun;$
$x \in Var;$
$lt \in VTerm(n);$
$lp \in ListPT$
$) \wedge(\subseteq(vars_{LPT}(:(lp, .(var(x), fun(f, lt)))), vars_{LPT}(:(lp, .(fun(f, lt), var(x))))),$
$\subseteq(vars_{LPT}(:(lp, .(fun(f, lt), var(x)))), vars_{LPT}(:(lp, .(var(x), fun(f, lt))))))$

$\subseteq vars_{var\_fun}(f, x, lt, lp) \equiv \wedge_I(\subseteq vars_{comm}(var(x), fun(f, lt), lp), \subseteq vars_{comm}(fun(f, lt), var(x), lp))$

$\subseteq vars_{zip\_++} \in (lt_1, lt_2 \in VTerm(n)) \subseteq(vars_{LPT}(zip(lt_1, lt_2)), +{+}(vars_{VT}(lt_1), vars_{VT}(lt_2)))$

$\subseteq vars_{zip\_++}([]_v, []_v) \equiv \subseteq_{[]}(+{+}(vars_{VT}([]_v), vars_{VT}([]_v)))$

$\subseteq vars_{zip\_++}(:_v(lt'_1, t_1), :_v(lt'_2, t_2)) \equiv$
$\subseteq_{trans}(\subseteq{+}{+}_{mon2R}(+{+}(vars_T(t_1), vars_T(t_2)), \subseteq vars_{zip\_++}(lt'_1, lt'_2)),$
$\subseteq_{assoc\_comm}(vars_{VT}(lt'_1), vars_{VT}(lt'_2), vars_T(t_1), vars_T(t_2)))$

$\subseteq vars_{++\_zip} \in (lt_1, lt_2 \in VTerm(n)) \subseteq(+{+}(vars_{VT}(lt_1), vars_{VT}(lt_2)), vars_{LPT}(zip(lt_1, lt_2)))$

$\subseteq vars_{++\_zip}([]_v, []_v) \equiv \subseteq_{[]}(vars_{LPT}(zip([]_v, []_v)))$

$\subseteq vars_{++\_zip}(:_v(lt'_1, t_1), :_v(lt'_2, t_2)) \equiv$
$\subseteq_{trans}(\subseteq_{assoc\_comm}(vars_{VT}(lt'_1), vars_T(t_1), vars_{VT}(lt'_2), vars_T(t_2)),$
$\subseteq{+}{+}_{mon2R}(+{+}(vars_T(t_1), vars_T(t_2)), \subseteq vars_{++\_zip}(lt'_1, lt'_2)))$

```
⊆vars++1 ∈ (lp1, lp2 ∈ ListPT) ⊆(vars_LPT(++(lp1, lp2)), ++(vars_LPT(lp1), vars_LPT(lp2)))
    ⊆vars++1(lp1, []) ≡ ⊆refl(vars_LPT(lp1))
    ⊆vars++1(lp1, :(lp'2, .(t1, t2))) ≡
        ⊆trans(⊆++mon2R(++(vars_T(t1), vars_T(t2)), ⊆vars++1(lp1, lp'2)),
               ⊆assoc1(vars_LPT(lp1), vars_LPT(lp'2), ++(vars_T(t1), vars_T(t2))))
⊆vars++2 ∈ (lp1, lp2 ∈ ListPT) ⊆(++(vars_LPT(lp1), vars_LPT(lp2)), vars_LPT(++(lp1, lp2)))
    ⊆vars++2(lp1, []) ≡ ⊆refl(vars_LPT(lp1))
    ⊆vars++2(lp1, :(lp'2, .(t1, t2))) ≡
        ⊆trans(⊆assoc2(vars_LPT(lp1), vars_LPT(lp'2), ++(vars_T(t1), vars_T(t2))),
               ⊆++mon2R(++(vars_T(t1), vars_T(t2)), ⊆vars++2(lp1, lp'2)))
⊆vars_fun_fun ∈ (f, g ∈ Fun;
                 lt1, lt2 ∈ VTerm(n);
                 lp ∈ ListPT
                ) ∧(⊆(vars_LPT(++(zip(lt1, lt2), lp)), vars_LPT(:(lp, .(fun(f, lt1), fun(g, lt2))))),
                    ⊆(vars_LPT(:(lp, .(fun(f, lt1), fun(g, lt2)))), vars_LPT(++(zip(lt1, lt2), lp))))
    ⊆vars_fun_fun(f, g, lt1, lt2, lp) ≡
        ∧I(⊆trans(⊆trans(⊆vars++1(zip(lt1, lt2), lp), ⊆++mon2R(vars_LPT(lp), ⊆vars_zip_++(lt1, lt2))),
                  ⊆comm(++(vars_VT(lt1), vars_VT(lt2)), vars_LPT(lp))),
           ⊆trans(⊆trans(⊆comm(vars_LPT(lp), ++(vars_VT(lt1), vars_VT(lt2))),
                         ⊆++mon2R(vars_LPT(lp), ⊆vars++_zip(lt1, lt2))),
                  ⊆vars++2(zip(lt1, lt2), lp)))
≤#vars_var_var ∈ (x ∈ Var; lp ∈ ListPT) ≤(#vars_LPT(lp), #vars_LPT(:(lp, .(var(x), var(x)))))
    ≤#vars_var_var(x, lp) ≡ ⊆to≤(⊆vars_var_var(x, lp))
<#vars:=_LPT ∈ (lp ∈ ListPT;
                ∉L(x, vars_T(t))
               ) <(#vars_LPT(:=_LPT(x, t, lp)), #vars_LPT(:(lp, .(var(x), t))))
    <#vars:=_LPT(lp, h) ≡
        ⊂to<(vars_LPT(:=_LPT(x, t, lp)),
             ∉to¬∈(∉:=_LPT=var(lp, h)),
             ∈var_term(x, t, lp),
             ⊆vars_var_term(x, t, lp))
=#vars_var_fun ∈ (f ∈ Fun;
                  x ∈ Var;
                  lt ∈ VTerm(n);
                  lp ∈ ListPT
                 ) =(#vars_LPT(:(lp, .(var(x), fun(f, lt)))), #vars_LPT(:(lp, .(fun(f, lt), var(x)))))
    =#vars_var_fun(f, x, lt, lp) ≡
        ≤∧≤to=(⊆to≤(fst(⊆vars_var_fun(f, x, lt, lp))), ⊆to≤(snd(⊆vars_var_fun(f, x, lt, lp))))
=#vars_fun_fun ∈ (f, g ∈ Fun;
                  lt1, lt2 ∈ VTerm(n);
                  lp ∈ ListPT
                 ) =(#vars_LPT(++(zip(lt1, lt2), lp)), #vars_LPT(:(lp, .(fun(f, lt1), fun(g, lt2)))))
    =#vars_fun_fun(f, g, lt1, lt2, lp) ≡
        ≤∧≤to=(⊆to≤(fst(⊆vars_fun_fun(f, g, lt1, lt2, lp))), ⊆to≤(snd(⊆vars_fun_fun(f, g, lt1, lt2, lp))))
=#funs_var_var ∈ (x ∈ Var; lp ∈ ListPT) =(#funs_LPT(lp), #funs_LPT(:(lp, .(var(x), var(x)))))
    =#funs_var_var(x, lp) ≡ refl(#funs_LPT(lp))
```

$=\#\mathrm{funs}_{\mathrm{var\_fun}} \in (f \in \mathrm{Fun};\ x \in \mathrm{Var};\ lt \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) = (\#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt)))), \#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$

$=\#\mathrm{funs}_{\mathrm{var\_fun}}(f, x, lt, lp) \equiv =_{\mathrm{cong1}}(+(\#\mathrm{funs}_{\mathrm{LPT}}(lp)), +_{\mathrm{comm}}(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{var}(x)), \#\mathrm{funs}_{\mathrm{T}}(\mathrm{fun}(f, lt))))$

$=\#\mathrm{funs}_{\mathrm{zip}} \in (lt_1, lt_2 \in \mathrm{VTerm}(n)) = (\#\mathrm{funs}_{\mathrm{LPT}}(\mathrm{zip}(lt_1, lt_2)), +(\#\mathrm{funs}_{\mathrm{VT}}(lt_1), \#\mathrm{funs}_{\mathrm{VT}}(lt_2)))$

$=\#\mathrm{funs}_{\mathrm{zip}}([]_{\mathrm{v}}, []_{\mathrm{v}}) \equiv \mathrm{refl}(\#\mathrm{funs}_{\mathrm{LPT}}([]))$

$=\#\mathrm{funs}_{\mathrm{zip}}(:_{\mathrm{v}}(lt'_1, t_1), :_{\mathrm{v}}(lt'_2, t_2)) \equiv$

$=_{\mathrm{trans}}(=_{\mathrm{cong1}}([n]+(n, +(\#\mathrm{funs}_{\mathrm{T}}(t_1), \#\mathrm{funs}_{\mathrm{T}}(t_2))), =\#\mathrm{funs}_{\mathrm{zip}}(lt'_1, lt'_2)),$

$+_{\mathrm{assoc\_comm}}(\#\mathrm{funs}_{\mathrm{VT}}(lt'_1), \#\mathrm{funs}_{\mathrm{VT}}(lt'_2), \#\mathrm{funs}_{\mathrm{T}}(t_1), \#\mathrm{funs}_{\mathrm{T}}(t_2)))$

$=\#\mathrm{funs}_{++} \in (lp_1, lp_2 \in \mathrm{ListPT}) = (\#\mathrm{funs}_{\mathrm{LPT}}(++(lp_1, lp_2)), +(\#\mathrm{funs}_{\mathrm{LPT}}(lp_1), \#\mathrm{funs}_{\mathrm{LPT}}(lp_2)))$

$=\#\mathrm{funs}_{++}(lp_1, []) \equiv \mathrm{refl}(\#\mathrm{funs}_{\mathrm{LPT}}(lp_1))$

$=\#\mathrm{funs}_{++}(lp_1, :(lp'_2, .(t_1, t_2))) \equiv$

$=_{\mathrm{trans}}(=_{\mathrm{cong1}}([n]+(n, +(\#\mathrm{funs}_{\mathrm{T}}(t_1), \#\mathrm{funs}_{\mathrm{T}}(t_2))), =\#\mathrm{funs}_{++}(lp_1, lp'_2)),$

$+_{\mathrm{assoc}}(\#\mathrm{funs}_{\mathrm{LPT}}(lp_1), \#\mathrm{funs}_{\mathrm{LPT}}(lp'_2), +(\#\mathrm{funs}_{\mathrm{T}}(t_1), \#\mathrm{funs}_{\mathrm{T}}(t_2))))$

$<\#\mathrm{funs}_{\mathrm{fun\_fun}} \in (f, g \in \mathrm{Fun};\ lt_1, lt_2 \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) < (\#\mathrm{funs}_{\mathrm{LPT}}(++(\mathrm{zip}(lt_1, lt_2), lp)), \#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(g, lt_2)))))$

$<\#\mathrm{funs}_{\mathrm{fun\_fun}}(f, g, lt_1, lt_2, lp) \equiv$

$=_{\mathrm{subst2}}(=\#\mathrm{funs}_{++}(\mathrm{zip}(lt_1, lt_2), lp),$

$+_{\mathrm{comm}}(+(\#\mathrm{funs}_{\mathrm{T}}(\mathrm{fun}(f, lt_1)), \#\mathrm{funs}_{\mathrm{T}}(\mathrm{fun}(g, lt_2))), \#\mathrm{funs}_{\mathrm{LPT}}(lp)),$

$<_{\mathrm{monR}}(\#\mathrm{funs}_{\mathrm{LPT}}(lp),$

$=_{\mathrm{subst2}}(=\#\mathrm{funs}_{\mathrm{zip}}(lt_1, lt_2),$

$=_{\mathrm{cong1}}(\mathrm{s}, =_{\mathrm{symm}}(\mathrm{s}+(\#\mathrm{funs}_{\mathrm{VT}}(lt_1), \#\mathrm{funs}_{\mathrm{VT}}(lt_2)))),$

$<_{\mathrm{ssR}}(+(\#\mathrm{funs}_{\mathrm{VT}}(lt_1), \#\mathrm{funs}_{\mathrm{VT}}(lt_2))))))$

$<\#\mathrm{eqs}_{\mathrm{var\_var}} \in (x \in \mathrm{Var};\ lp \in \mathrm{ListPT}) < (\#\mathrm{eqs}_{\mathrm{LPT}}(lp), \#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$

$<\#\mathrm{eqs}_{\mathrm{var\_var}}(x, lp) \equiv$

$=_{\mathrm{subst1}}(=\mathrm{var}_{\mathrm{to}}\mathrm{A}(+(\#\mathrm{eqs}_{\mathrm{LPT}}(lp), 1), \#\mathrm{eqs}_{\mathrm{LPT}}(lp), \mathrm{Var}_{\mathrm{dec}}(x, x)), <_{\mathrm{sR}}(\#\mathrm{eqs}_{\mathrm{LPT}}(lp)))$

$<\#\mathrm{eqs}_{\mathrm{var\_fun}} \in (f \in \mathrm{Fun};\ x \in \mathrm{Var};\ lt \in \mathrm{VTerm}(n);\ lp \in \mathrm{ListPT}) < (\#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt)))), \#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$

$<\#\mathrm{eqs}_{\mathrm{var\_fun}}(f, x, lt, lp) \equiv <_{\mathrm{sR}}(\#\mathrm{eqs}_{\mathrm{LPT}}(lp))$

$<_{\mathrm{LPTvar\_var}} \in (x \in \mathrm{Var};\ lp \in \mathrm{ListPT}) <_{\mathrm{N3}}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(lp), \mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$

$<_{\mathrm{LPTvar\_var}}(x, lp) \equiv <_{\mathrm{fst\_rd'}}(\leq\#\mathrm{vars}_{\mathrm{var\_var}}(x, lp), =\#\mathrm{funs}_{\mathrm{var\_var}}(x, lp), <\#\mathrm{eqs}_{\mathrm{var\_var}}(x, lp))$

$<_{\mathrm{LPT:=var\_term}} \in (lp \in \mathrm{ListPT};\ \notin_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t))) <_{\mathrm{N3}}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:=_{\mathrm{LPT}}(x, t, lp)), \mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:(lp, .(\mathrm{var}(x), t))))$

$<_{\mathrm{LPT:=var\_term}}(lp, h) \equiv$

$<_{\mathrm{fst}}(\#\mathrm{funs}_{\mathrm{LPT}}(:=_{\mathrm{LPT}}(x, t, lp)),$

$\#\mathrm{funs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), t))),$

$\#\mathrm{eqs}_{\mathrm{LPT}}(:=_{\mathrm{LPT}}(x, t, lp)),$

$\#\mathrm{eqs}_{\mathrm{LPT}}(:(lp, .(\mathrm{var}(x), t))),$

$<\#\mathrm{vars}_{:=\mathrm{LPT}}(lp, h))$

$<_{LPTvar\_fun} \in (f \in Fun;$
$x \in Var;$
$lt \in VTerm(n);$
$lp \in ListPT$
$) <_{N3}(LPT_{to}N3(:(lp, .(var(x), fun(f, lt)))), LPT_{to}N3(:(lp, .(fun(f, lt), var(x)))))$

$<_{LPTvar\_fun}(f, x, lt, lp) \equiv$
$<_{rd'}(=\#vars_{var\_fun}(f, x, lt, lp), =\#funs_{var\_fun}(f, x, lt, lp), <\#eqs_{var\_fun}(f, x, lt, lp))$

$<_{LPTzip\_fun\_fun} \in (f, g \in Fun;$
$lt_1, lt_2 \in VTerm(n);$
$lp \in ListPT$
$) <_{N3}(LPT_{to}N3(++(zip(lt_1, lt_2), lp)), LPT_{to}N3(:(lp, .(fun(f, lt_1), fun(g, lt_2)))))$

$<_{LPTzip\_fun\_fun}(f, g, lt_1, lt_2, lp) \equiv$
$<_{snd'}(\#eqs_{LPT}(++(zip(lt_1, lt_2), lp)),$
$\#eqs_{LPT}(:(lp, .(fun(f, lt_1), fun(g, lt_2)))),$
$=\#vars_{fun\_fun}(f, g, lt_1, lt_2, lp),$
$<\#funs_{fun\_fun}(f, g, lt_1, lt_2, lp))$

$unifies_{LPTto=_f} \in (unifies_{LPT}(sb, :(lp, .(fun(f, lt_1), fun(g, lt_2)))) =(f, g)$
$unifies_{LPTto=_f}(unifies_{LPT\_:}(h_1, h_2)) \equiv =_{funto=_f}(h_2)$

$unifies_{LPTto=_{arity}} \in (unifies_{LPT}(sb, :(lp, .(fun(f, lt_1), fun(g, lt_2)))) =(n_1, n_2)$
$unifies_{LPTto=_{arity}}(unifies_{LPT\_:}(h_1, h_2)) \equiv =_{funto=_{arity}}(h_2)$

$unifies_{LPT\_red} \in (unifies_{LPT}(sb, :(lp, .(var(x), var(x))))) unifies_{LPT}(sb, lp)$
$unifies_{LPT\_red}(unifies_{LPT\_:}(h_1, h_2)) \equiv h_1$

$unifies_{LPT\wedge\in_{to}\perp} \in (unifies_{LPT}(sb, :(lp, .(var(x), t))); \in_L(x, vars_T(t)); \neg(=(var(x), t))) \perp$
$unifies_{LPT\wedge\in_{to}\perp}(unifies_{LPT\_:}(h_3, h_4), h_1, h_2) \equiv \in\wedge\neq\wedge unify_{to}\perp(t, h_1, h_2, h_4)$

$unifies_{LPT\_:=_L} \in (lp \in ListPT;$
$=(appP_T(sb, var(x)), appP_T(sb, t));$
$unifies_{LPT}(sb, :=_{LPT}(x, t, lp))$
$) unifies_{LPT}(sb, lp)$

$unifies_{LPT\_:=_L}([], h, h_1) \equiv unifies_{LPT\_[]}(sb)$
$unifies_{LPT\_:=_L}(:(lp', .(t_1, t_2)), h, unifies_{LPT\_:}(h_2, h_3)) \equiv$
$unifies_{LPT\_:}(unifies_{LPT\_:=_L}(lp', h, h_2), =_{subst2}(=:=_{appP\_T}(t_1, h), =_{symm}(=:=_{appP\_T}(t_2, h)), h_3))$

$unifies_{LPT\_:=_{R\_aux}} \in (lp \in ListPT;$
$=(appP_T(sb, var(x)), appP_T(sb, t));$
$unifies_{LPT}(sb, lp)$
$) unifies_{LPT}(sb, :=_{LPT}(x, t, lp))$

$unifies_{LPT\_:=_{R\_aux}}([], h, h_1) \equiv unifies_{LPT\_[]}(sb)$
$unifies_{LPT\_:=_{R\_aux}}(:(lp', .(t_1, t_2)), h, unifies_{LPT\_:}(h_2, h_3)) \equiv$
$unifies_{LPT\_:}(unifies_{LPT\_:=_{R\_aux}}(lp', h, h_2),$
$=_{subst2}(=_{symm}(=:=_{appP\_T}(t_1, h)), =:=_{appP\_T}(t_2, h), h_3))$

$unifies_{LPT\_:=_R} \in (unifies_{LPT}(sb, :(lp, .(var(x), t)))) unifies_{LPT}(sb, :=_{LPT}(x, t, lp))$
$unifies_{LPT\_:=_R}(unifies_{LPT\_:}(h_1, h_2)) \equiv unifies_{LPT\_:=_{R\_aux}}(lp, h_2, h_1)$

$unifies_{LPT\_vf_{to}fv} \in (unifies_{LPT}(sb, :(lp, .(var(x), fun(f, lt))))$
$) unifies_{LPT}(sb, :(lp, .(fun(f, lt), var(x))))$
$unifies_{LPT\_vf_{to}fv}(unifies_{LPT\_:}(h_1, h_2)) \equiv unifies_{LPT\_:}(h_1, =_{symm}(h_2))$

$unifies_{LPT\_fv_{to}vf} \in (unifies_{LPT}(sb, :(lp, .(fun(f, lt), var(x))))$
$) unifies_{LPT}(sb, :(lp, .(var(x), fun(f, lt))))$
$unifies_{LPT\_fv_{to}vf}(unifies_{LPT\_:}(h_1, h_2)) \equiv unifies_{LPT\_:}(h_1, =_{symm}(h_2))$

$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}L}} \in (lp_2 \in \mathrm{ListPT};$
$\mathrm{unifies}_{\mathrm{LPT}}(sb, {+}{+}(lp_1, lp_2))$
$) \wedge (\mathrm{unifies}_{\mathrm{LPT}}(sb, lp_1), \mathrm{unifies}_{\mathrm{LPT}}(sb, lp_2))$

$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}L}}([], h) \equiv \wedge_{\mathrm{I}}(h, \mathrm{unifies}_{\mathrm{LPT\_[]}}(sb))$

$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}L}}(:(lp, \_), \mathrm{unifies}_{\mathrm{LPT\_:}}(h_1, h_2)) \equiv$
case $\mathrm{unifies}_{\mathrm{LPT\_{+}{+}L}}(lp, h_1) \in \wedge(\mathrm{unifies}_{\mathrm{LPT}}(sb, lp_1), \mathrm{unifies}_{\mathrm{LPT}}(sb, lp))$ **of**
$\wedge_{\mathrm{I}}(h, h_3) \Rightarrow \wedge_{\mathrm{I}}(h, \mathrm{unifies}_{\mathrm{LPT\_:}}(h_3, h_2))$
end

$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}R}} \in (lp_2 \in \mathrm{ListPT};$
$\mathrm{unifies}_{\mathrm{LPT}}(sb, lp_1);$
$\mathrm{unifies}_{\mathrm{LPT}}(sb, lp_2)$
$) \mathrm{unifies}_{\mathrm{LPT}}(sb, {+}{+}(lp_1, lp_2))$

$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}R}}([], h, h_1) \equiv h$

$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}R}}(:(lp, .(t_1, t_2)), h, \mathrm{unifies}_{\mathrm{LPT\_:}}(h_2, h_3)) \equiv$
$\mathrm{unifies}_{\mathrm{LPT\_:}}(\mathrm{unifies}_{\mathrm{LPT\_{+}{+}R}}(lp, h, h_2), h_3)$

$\mathrm{unifies}_{\mathrm{LPT\_zipL}} \in (lt_1, lt_2 \in \mathrm{VTerm}(n);$
$\mathrm{unifies}_{\mathrm{LPT}}(sb, \mathrm{zip}(lt_1, lt_2))$
$) = (\mathrm{appP}_{\mathrm{VT}}(sb, lt_1), \mathrm{appP}_{\mathrm{VT}}(sb, lt_2))$

$\mathrm{unifies}_{\mathrm{LPT\_zipL}}([]_{\mathrm{v}}, []_{\mathrm{v}}, h) \equiv \mathrm{refl}(\mathrm{appP}_{\mathrm{VT}}(sb, []_{\mathrm{v}}))$

$\mathrm{unifies}_{\mathrm{LPT\_zipL}}(:_{\mathrm{v}}(lt'_1, t_1), :_{\mathrm{v}}(lt'_2, t_2), \mathrm{unifies}_{\mathrm{LPT\_:}}(h_1, h_2)) \equiv$
$=_{\mathrm{cong2}}(:_{\mathrm{v}}, \mathrm{unifies}_{\mathrm{LPT\_zipL}}(lt'_1, lt'_2, h_1), h_2)$

$\mathrm{unifies}_{\mathrm{LPT\_zipR}} \in (lt_1, lt_2 \in \mathrm{VTerm}(n);$
$= (\mathrm{appP}_{\mathrm{VT}}(sb, lt_1), \mathrm{appP}_{\mathrm{VT}}(sb, lt_2))$
$) \mathrm{unifies}_{\mathrm{LPT}}(sb, \mathrm{zip}(lt_1, lt_2))$

$\mathrm{unifies}_{\mathrm{LPT\_zipR}}([]_{\mathrm{v}}, []_{\mathrm{v}}, h) \equiv \mathrm{unifies}_{\mathrm{LPT\_[]}}(sb)$

$\mathrm{unifies}_{\mathrm{LPT\_zipR}}(:_{\mathrm{v}}(lt'_1, t_1), :_{\mathrm{v}}(lt'_2, t_2), h) \equiv$
$\mathrm{unifies}_{\mathrm{LPT\_:}}(\mathrm{unifies}_{\mathrm{LPT\_zipR}}(lt'_1, lt'_2, =_{\mathrm{vector\,to{=}tail}}(h)), =_{\mathrm{vector\,to{=}head}}(h))$

$\mathrm{unifies}_{\mathrm{LPT\_fun_{to}zip}} \in (\mathrm{unifies}_{\mathrm{LPT}}(sb, :(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(f, lt_2)))))$
$) \mathrm{unifies}_{\mathrm{LPT}}(sb, {+}{+}(\mathrm{zip}(lt_1, lt_2), lp))$

$\mathrm{unifies}_{\mathrm{LPT\_fun_{to}zip}}(\mathrm{unifies}_{\mathrm{LPT\_:}}(h_1, h_2)) \equiv$
$\mathrm{unifies}_{\mathrm{LPT\_{+}{+}R}}(lp, \mathrm{unifies}_{\mathrm{LPT\_zipR}}(lt_1, lt_2, =_{\mathrm{fun\,to{=}vector}}(h_2)), h_1)$

$\mathrm{unifies}_{\mathrm{LPT\_zip_{to}fun}} \in (f \in \mathrm{Fun};$
$\mathrm{unifies}_{\mathrm{LPT}}(sb, {+}{+}(\mathrm{zip}(lt_1, lt_2), lp))$
$) \mathrm{unifies}_{\mathrm{LPT}}(sb, :(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(f, lt_2)))))$

$\mathrm{unifies}_{\mathrm{LPT\_zip_{to}fun}}(f, h) \equiv$
case $\mathrm{unifies}_{\mathrm{LPT\_{+}{+}L}}(lp, h) \in \wedge(\mathrm{unifies}_{\mathrm{LPT}}(sb, \mathrm{zip}(lt_1, lt_2)), \mathrm{unifies}_{\mathrm{LPT}}(sb, lp))$ **of**
$\wedge_{\mathrm{I}}(h_1, h_2) \Rightarrow \mathrm{unifies}_{\mathrm{LPT\_:}}(h_2, =_{\mathrm{cong1}}(\mathrm{fun}(f), \mathrm{unifies}_{\mathrm{LPT\_zipL}}(lt_1, lt_2, h_1)))$
end

$\mathrm{unifies}_{\mathrm{LPT}}\mathrm{from}_{\mathrm{sb\,to\,sb:}} \in (lp \in \mathrm{ListPT};$
$\notin_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{LPT}}(lp));$
$\mathrm{unifies}_{\mathrm{LPT}}(sb, lp);$
$\mathrm{Idem}(:(sb, .(x, t)))$
$) \mathrm{unifies}_{\mathrm{LPT}}(:(sb, .(x, t)), lp)$

$\mathrm{unifies}_{\mathrm{LPT}}\mathrm{from}_{\mathrm{sb\,to\,sb:}}([], h, h_1, h_2) \equiv \mathrm{unifies}_{\mathrm{LPT\_[]}}(:(sb, .(x, t)))$

$\mathrm{unifies}_{\mathrm{LPT}}\mathrm{from}_{\mathrm{sb\,to\,sb:}}(:(lp', .(t_1, t_2)), h, \mathrm{unifies}_{\mathrm{LPT\_:}}(h_3, h_4), h_2) \equiv$
$\mathrm{unifies}_{\mathrm{LPT\_:}}($
$\mathrm{unifies}_{\mathrm{LPT}}\mathrm{from}_{\mathrm{sb\,to\,sb:}}(lp', \notin {+}{+}_{\mathrm{redR}}({+}{+}(\mathrm{vars}_{\mathrm{T}}(t_1), \mathrm{vars}_{\mathrm{T}}(t_2)), h), h_3, h_2),$
$=_{\mathrm{subst2}}(\notin_{\mathrm{to}}{=}_{\mathrm{T\_app^P}}(\notin {+}{+}_{\mathrm{redR}}(\mathrm{vars}_{\mathrm{T}}(t_2), \notin {+}{+}_{\mathrm{redL}}({+}{+}(\mathrm{vars}_{\mathrm{T}}(t_1), \mathrm{vars}_{\mathrm{T}}(t_2)), h)), h_2),$
$=_{\mathrm{symm}}(\notin_{\mathrm{to}}{=}_{\mathrm{T\_app^P}}(\notin {+}{+}_{\mathrm{redL}}(\mathrm{vars}_{\mathrm{T}}(t_2), \notin {+}{+}_{\mathrm{redL}}({+}{+}(\mathrm{vars}_{\mathrm{T}}(t_1), \mathrm{vars}_{\mathrm{T}}(t_2)), h)), h_2)),$
$h_4))$

```
unifies_LPT:=_to unifies_LPT ∈ (∉_L(x, vars_T(t));
                                ∉_L(x, vars_S(sb));
                                Idem(sb);
                                unifies_LPT(sb, :=_LPT(x, t, lp))
                               ) unifies_LPT(:(sb, .(x, appP_T(sb, t))), lp)
    unifies_LPT:=_to unifies_LPT(h, h1, h2, h3) ≡
        unifies_LPT_:=_L(lp,
                         =_var_term appP_T(h, h1, h2),
                         unifies_LPT from_sb to_sb:(:=_LPT(x, t, lp), ∉:=_LPT=_var(lp, h), h3, idem:(h, h1, h2)))
```

C.4.5 Substitutions

```
PairS ∈ Set
    PairS ≡ Pair(Var, Term)
Subst ∈ Set
    Subst ≡ List(PairS)
dom ∈ (sb ∈ Subst) ListVar
    dom([]) ≡ []
    dom(:(sb', .(x, t))) ≡ :(dom(sb'), x)
vars_S ∈ (sb ∈ Subst) ListVar
    vars_S([]) ≡ []
    vars_S(:(sb', .(x, t))) ≡ ++(vars_S(sb'), :(vars_T(t), x))
Idem ∈ (sb ∈ Subst) Set
    []_idem ∈ Idem([])
    :_idem ∈ (Idem(sb);
              Disjoint(vars_T(t), dom(:(sb, .(x, t))));
              ∉_L(x, vars_S(sb))
             ) Idem(:(sb, .(x, t)))
:=_S ∈ (x ∈ Var; t ∈ Term; sb ∈ Subst) Subst
    :=_S(x, t, []) ≡ []
    :=_S(x, t, :(sb1, .(y, t1))) ≡ :(:=_S(x, t, sb1), .(y, :=_T(x, t, t1)))
appP_T ∈ (sb ∈ Subst; t ∈ Term) Term
    appP_T([], var(x)) ≡ var(x)
    appP_T(:(sb1, .(x1, t)), var(x)) ≡ Var_to A(t, appP_T(sb1, var(x)), Var_dec(x1, x))
    appP_T(sb, fun(f, lt)) ≡ fun(f, appP_VT(sb, lt))
appP_VT ∈ (sb ∈ Subst; lt ∈ VTerm(n)) VTerm(n)
    appP_VT(sb, []_v) ≡ []_v
    appP_VT(sb, :_v(lt', t)) ≡ :_v(appP_VT(sb, lt'), appP_T(sb, t))
appS_T ∈ (sb ∈ Subst; t ∈ Term) Term
    appS_T([], t) ≡ t
    appS_T(:(sb1, .(x, t1)), t) ≡ appS_T(sb1, :=_T(x, t1, t))
appS_VT ∈ (sb ∈ Subst; lt ∈ VTerm(n)) VTerm(n)
    appS_VT(sb, []_v) ≡ []_v
    appS_VT(sb, :_v(lt', t)) ≡ :_v(appS_VT(sb, lt'), appS_T(sb, t))
≤_Sb ∈ (sb, sb' ∈ Subst) Set
    ≤_Sb ≡ [sb, sb']∃(Subst, [sb1]∀(Term, [t]=(appP_T(sb', t), appP_T(sb1, appP_T(sb, t)))))
Idempotent ∈ (sb ∈ Subst) Set
    Idempotent ≡ [sb]∀(Term, [t]=(appP_T(sb, appP_T(sb, t)), appP_T(sb, t)))
```

$=_{var}appP \in (x \in Var; t \in Term; sb \in Subst) =(t, appP_T(:(sb, .(x, t)), var(x)))$
$=_{var}appP(x, t, sb) \equiv =var_{to}A(t, appP_T(sb, var(x)), Var_{dec}(x, x))$
$\neq_{var}appP \in (t \in Term;$
$sb \in Subst;$
$\neg(=(y, x))$
$) =(appP_T(sb, var(x)), appP_T(:(sb, .(y, t)), var(x)))$
$\neq_{var}appP(t, sb, h) \equiv \neq var_{to}A(t, appP_T(sb, var(x)), h, Var_{dec}(y, x))$
$unifies_S \in (sb_1, sb_2 \in Subst)$ **Set**
$unifies_{S\_[]} \in (sb \in Subst)\ unifies_S(sb, [])$
$unifies_{S\_:} \in (unifies_S(sb, sb_1);$
$=(appP_T(sb, var(x)), appP_T(sb, t))$
$) unifies_S(sb, :(sb_1, .(x, t)))$

C.4.6 Properties of Substitutions

$=_{dom} \in (x \in Var; t \in Term; sb \in Subst) =(dom(sb), dom(:=_S(x, t, sb)))$
$=_{dom}(x, t, []) \equiv refl(dom([]))$
$=_{dom}(x, t, :(sb_1, .(y, t_1))) \equiv =_{cong2}(:, =_{dom}(x, t, sb_1), refl(y))$
$\in_{to}\leq_{\#funsT} \in (sb \in Subst;$
$t \in Term;$
$\in_L(x, vars_T(t))$
$) \leq(\#funs_T(appP_T(sb, var(x))), \#funs_T(appP_T(sb, t)))$
$\in_{to}\leq_{\#funsT}(sb, var(x_1), \in_{hd}(\_, \_)) \equiv \vee_R(refl(\#funs_T(appP_T(sb, var(x_1)))))$
$\in_{to}\leq_{\#funsT}(sb, var(x_1), \in_{tl}(\_, \_, h_1)) \equiv$ **case** $h_1 \in \in_L(x, [])$ **of**
end
$\in_{to}\leq_{\#funsT}(sb, fun(f, lt), h) \equiv \leq_{to}\leq_{+R}(s(0), \in_{to}\leq_{\#funsVT}(sb, lt, h))$
$\in_{to}\leq_{\#funsVT} \in (sb \in Subst;$
$lt \in VTerm(n);$
$\in_L(x, vars_{VT}(lt))$
$) \leq(\#funs_T(appP_T(sb, var(x))), \#funs_{VT}(appP_{VT}(sb, lt)))$
$\in_{to}\leq_{\#funsVT}(sb, []_v, h) \equiv$ **case** $h \in \in_L(x, vars_{VT}([]_v))$ **of**
end
$\in_{to}\leq_{\#funsVT}(sb, :_v(lt', t'), h) \equiv$
case $\in ++_{to}\in \vee(vars_T(t'), h) \in \vee(\in_L(x, vars_{VT}(lt')), \in_L(x, vars_T(t')))$ **of**
$\vee_L(h_1) \Rightarrow \leq_{to}\leq_{+R}(\#funs_T(appP_T(sb, t')), \in_{to}\leq_{\#funsVT}(sb, lt', h_1))$
$\vee_R(h_2) \Rightarrow$
$=_{subst1}(+_{comm}(\#funs_T(appP_T(sb, t')), \#funs_{VT}(appP_{VT}(sb, lt'))),$
$\leq_{to}\leq_{+R}(\#funs_{VT}(appP_{VT}(sb, lt')), \in_{to}\leq_{\#funsT}(sb, t', h_2)))$
end
$\notin_{vars to}\notin_{dom} \in (sb \in Subst; \notin_L(x, vars_S(sb))) \notin_L(x, dom(sb))$
$\notin_{vars to}\notin_{dom}([], h) \equiv \notin_{[]}(x)$
$\notin_{vars to}\notin_{dom}(:(sb_1, .(y, t)), \notin_{:}(h_1, h_2)) \equiv \notin_{:}(\notin_{vars to}\notin_{dom}(sb_1, \notin ++_{redR}(vars_T(t), h_1)), h_2)$
$\notin_{vars\_appS} \in (sb \in Subst; \notin_L(x, vars_T(t)); \notin_L(x, vars_S(sb))) \notin_L(x, vars_T(appS_T(sb, t)))$
$\notin_{vars\_appS}([], h, h_1) \equiv h$
$\notin_{vars\_appS}(:(sb_1, .(x_1, t_1)), h, \notin_{:}(h_2, h_3)) \equiv$
$\notin_{vars\_appS}(sb_1, \notin :=_T\neq_{var}(x_1, t, \notin ++_{redL}(vars_T(t_1), h_2), h), \notin ++_{redR}(vars_T(t_1), h_2))$

```
∉:=S=var ∈ (sb ∈ Subst; ∉L(x, varsT(t)); ∉L(x, dom(sb))) ∉L(x, varsS(:=S(x, t, sb)))
    ∉:=S=var([], h, h1) ≡ ∉[](x)
    ∉:=S=var(:(sb1, .(y, t1)), h, ∉:(h2, h3)) ≡ ∉++(∉:=S=var(sb1, h, h2), ∉:(∉:=T=var(t1, h), h3))
∉:=S≠var ∈ (y ∈ Var; sb ∈ Subst; ∉L(x, varsT(t)); ∉L(x, varsS(sb))) ∉L(x, varsS(:=S(y, t, sb)))
    ∉:=S≠var(y, [], h, h1) ≡ ∉[](x)
    ∉:=S≠var(y, :(sb1, .(z, t1)), h, ∉:(h2, h3)) ≡
        ∉++(∉:=S≠var(y, sb1, h, ∉++redR(varsT(t1), h2)),
            ∉:(∉:=T≠var(y, t1, h, ∉++redL(varsT(t1), h2)), h3))
:=Tdisj:=S ∈ (x ∈ Var;
              sb ∈ Subst;
              Disjoint(varsT(t), dom(sb));
              Disjoint(varsT(t1), dom(sb))
             ) Disjoint(varsT(:=T(x, t, t1)), dom(:=S(x, t, sb)))
    :=Tdisj:=S(x, [], h, h1) ≡ disj[]R(varsT(:=T(x, t, t1)))
    :=Tdisj:=S(x, :(sb1, .(y, t')), h, h1) ≡
        disj:R(:=Tdisj:=S(x, sb1, disjredR(h), disjredR(h1)), ∉:=T≠var(x, t1, disjto∉(h), disjto∉(h1)))
⊆vars:=S ∈ (x ∈ Var; sb ∈ Subst; ⊆(varsT(t), l); ⊆(varsS(sb), l)) ⊆(varsS(:=S(x, t, sb)), l)
    ⊆vars:=S(x, [], h, h1) ≡ ⊆[](l)
    ⊆vars:=S(x, :(sb1, .(y, t1)), h, ⊆:(h2, h3)) ≡
        ⊆:(⊆++L(⊆vars:=S(x, sb1, h, ⊆++redR(varsT(t1), h2)), ⊆vars:=T(x, t1, h, ⊆++redL(h2))), h3)
⊆varsappP_T ∈ (sb ∈ Subst; t ∈ Term) ⊆(varsT(appPT(sb, t)), ++(varsS(sb), varsT(t)))
    ⊆varsappP_T([], var(x)) ≡ ⊆comm(varsS([]), varsT(var(x)))
    ⊆varsappP_T(:(sb1, .(y, t1)), var(x)) ≡
        case Vardec(y, x) ∈ Dec(=(y, x)) of
            yes(refl(_)) ⇒
                ⊆++monR(varsT(var(x)),
                        =subst1(=varappP(x, t1, sb1),
                                ⊆++monL(varsS(sb1), ⊆:monR(x, ⊆refl(varsT(t1))))))
            no(h) ⇒
                ⊆trans(=subst1(≠varappP(t1, sb1, h), ⊆varsappP_T(sb1, var(x))),
                       ⊆++C(varsS(sb1), varsT(var(x)), :(varsT(t1), y)))
        end
    ⊆varsappP_T(sb, fun(f, lt)) ≡ ⊆varsappP_VT(sb, lt)
⊆varsappP_VT ∈ (sb ∈ Subst;
                lt ∈ VTerm(n)
               ) ⊆(varsVT(appPVT(sb, lt)), ++(varsS(sb), varsVT(lt)))
    ⊆varsappP_VT(sb, []v) ≡ ⊆[](varsS(sb))
    ⊆varsappP_VT(sb, :v(lt', t)) ≡
        ⊆trans(⊆++L(⊆++monR(++(varsS(sb), varsT(t)), ⊆varsappP_VT(sb, lt')),
                    ⊆++monL(++(varsS(sb), varsVT(lt')), ⊆varsappP_T(sb, t))),
               ⊆trans(⊆assoc_comm(varsS(sb), varsVT(lt'), varsS(sb), varsT(t)),
                      ⊆++mon2R(varsVT(:v(lt', t)), ⊆++refl(varsS(sb)))))
```

$=_{:=appP\_T} \in (t' \in \text{Term};$
$=(appP_T(sb, \text{var}(x)), appP_T(sb, t))$
$) =(appP_T(sb, t'), appP_T(sb, :=_T(x, t, t')))$
$=_{:=appP\_T}(\text{var}(x_1), h) \equiv \textbf{case } \text{Var}_{dec}(x, x_1) \in \text{Dec}(=(x, x_1)) \textbf{ of}$
$\text{yes}(\text{refl}(\_)) \Rightarrow =_{trans}(h, =_{cong1}(appP_T(sb), :=_{var}(x_1, t)))$
$\text{no}(h_1) \Rightarrow =_{cong1}(appP_T(sb), :=_{\neq var}(t, h_1))$
$\textbf{end}$
$=_{:=appP\_T}(\text{fun}(f, lt), h) \equiv =_{cong1}(\text{fun}(f), =_{:=appP\_VT}(lt, h))$
$=_{:=appP\_VT} \in (lt \in \text{VTerm}(n);$
$=(appP_T(sb, \text{var}(x)), appP_T(sb, t))$
$) =(appP_{VT}(sb, lt), appP_{VT}(sb, :=_{VT}(x, t, lt)))$
$=_{:=appP\_VT}([]_v, h) \equiv \text{refl}(appP_{VT}(sb, []_v))$
$=_{:=appP\_VT}(:_v(lt', t'), h) \equiv =_{cong2}(:_v, =_{:=appP\_VT}(lt', h), =_{:=appP\_T}(t', h))$
$=_{T disj\_vars} \in (sb \in \text{Subst}; \text{Disjoint}(\text{vars}_T(t), \text{dom}(sb))) =(t, appS_T(sb, t))$
$=_{T disj\_vars}([], h) \equiv \text{refl}(t)$
$=_{T disj\_vars}(:(sb', .(x, t_1)), h) \equiv$
$=_{trans}(=_{T disj\_vars}(sb', disj_{redR}(h)), =_{cong1}(appS_T(sb'), =_{:=T}(t_1, t, disj_{to\notin}(h))))$
$\text{disj}_{vars\_appS} \in (t \in \text{Term}; \text{Idem}(sb)) \text{Disjoint}(\text{vars}_T(appS_T(sb, t)), \text{dom}(sb))$
$\text{disj}_{vars\_appS}(t, []_{idem}) \equiv \text{disj}_{[]R}(\text{vars}_T(t))$
$\text{disj}_{vars\_appS}(t, :_{idem}(h_1, h_2, h_3)) \equiv$
$\text{disj}_{:R}(\text{disj}_{vars\_appS}(:=_T(x, t_1, t), h_1), \notin_{vars\_appS}(sb_1, \notin :=_T =_{var}(t, \text{disj}_{to\notin}(h_2)), h_3))$
$=_{VT}appS_{[]s} \in (lt \in \text{VTerm}(n)) =(appS_{VT}([], lt), lt)$
$=_{VT}appS_{[]s}([]_v) \equiv \text{refl}([]_v)$
$=_{VT}appS_{[]s}(:_v(lt', t)) \equiv =_{cong1}([lt_1]:_v(lt_1, t), =_{VT}appS_{[]s}(lt'))$
$=_{VT}appS_{.s} \in (x \in \text{Var};$
$t \in \text{Term};$
$sb \in \text{Subst};$
$lt \in \text{VTerm}(n)$
$) =(appS_{VT}(:(sb, .(x, t)), lt), appS_{VT}(sb, :=_{VT}(x, t, lt)))$
$=_{VT}appS_{.s}(x, t, sb, []_v) \equiv \text{refl}([]_v)$
$=_{VT}appS_{.s}(x, t, sb, :_v(lt', t')) \equiv$
$=_{cong1}([lt_1]:_v(lt_1, appS_T(:(sb, .(x, t)), t')), =_{VT}appS_{.s}(x, t, sb, lt'))$
$=_{VT}appS \in (f \in \text{Fun};$
$sb \in \text{Subst};$
$lt \in \text{VTerm}(n)$
$) =(\text{fun}(f, appS_{VT}(sb, lt)), appS_T(sb, \text{fun}(f, lt)))$
$=_{VT}appS(f, [], lt) \equiv =_{cong1}(\text{fun}(f), =_{VT}appS_{[]s}(lt))$
$=_{VT}appS(f, :(sb_1, .(x, t)), lt) \equiv$
$=_{trans}(=_{cong1}(\text{fun}(f), =_{VT}appS_{.s}(x, t, sb_1, lt)), =_{VT}appS(f, sb_1, :=_{VT}(x, t, lt)))$

```
=Tapp̄P–S ∈ (sb ∈ Subst; t ∈ Term; Idem(sb)) =(appPT(sb, t), appST(sb, t))
    =Tapp̄P–S([], var(x), h) ≡ refl(var(x))
    =Tapp̄P–S(:(sb1, .(x1, t)), var(x), :idem(h1, h2, h3)) ≡
        case Vardec(x1, x) ∈ Dec(=(x1, x)) of
            yes(refl(_)) ⇒
                =trans(=symm(=varappP(x, t, sb1)),
                    =trans(=Tdisj_vars(sb1, disjredR(h2)), =cong1(appST(sb1), :=̄=var(x, t))))
            no(h) ⇒
                =trans(=symm(≠varappP(t, sb1, h)),
                    =trans(=Tapp̄P–S(sb1, var(x), h1), =cong1(appST(sb1), :=̄≠var(t, h))))
        end
    =Tapp̄P–S(sb, fun(f, lt), h) ≡ =trans(=cong1(fun(f), =VTapp̄P–S(lt, h)), =VTappS(f, sb, lt))
=VTapp̄P–S ∈ (lt ∈ VTerm(n); Idem(sb)) =(appPVT(sb, lt), appSVT(sb, lt))
    =VTapp̄P–S([]v, h) ≡ refl([]v)
    =VTapp̄P–S(:v(lt', t), h) ≡ =cong2(:v, =VTapp̄P–S(lt', h), =Tapp̄P–S(sb, t, h))
idem∧disjto= ∈ (sb' ∈ Subst;
                Idem(sb);
                Disjoint(varsT(t), dom(sb))
               ) =(appPT(sb', t), appPT(sb', appPT(sb, t)))
    idem∧disjto=(sb', h, h1) ≡
        =cong1(appPT(sb'), =trans(=Tdisj_vars(sb, h1), =symm(=Tapp̄P–S(sb, t, h))))
idemtoidempotent ∈ (Idem(sb)) Idempotent(sb)
    idemtoidempotent(h) ≡
        ∀I([t]=subst2(=trans(=cong1(appPT(sb), =Tapp̄P–S(sb, t, h)), =Tapp̄P–S(sb, appST(sb, t), h)),
                =symm(=Tapp̄P–S(sb, t, h)),
                =symm(=Tdisj_vars(sb, disjvars_appS(t, h)))))
∉to=T_appP ∈ (∉L(x, varsT(t)); Idem(:(sb, .(x, t')))) =(appPT(:(sb, .(x, t')), t), appPT(sb, t))
    ∉to=T_appP(h, :idem(h2, h3, h4)) ≡
        =subst2(=Tapp̄P–S(:(sb, .(x, t')), t, :idem(h2, h3, h4)),
            =symm(=Tapp̄P–S(sb, t, h2)),
            =cong1(appST(sb), =symm(=:=T(t', t, h))))
```

```
=∧idem_to=T ∈ (t' ∈ Term;
                Idem(:(sb', .(x, t)));
                =(appP_T(sb, var(x)), appP_T(sb, t))
               ) =(appP_T(sb, appP_T(sb', t')), appP_T(sb, appP_T(:(sb', .(x, t)), t')))
  =∧idem_to=T(var(x1), :idem(h2, h3, h4), h1) ≡
    case Var_dec(x, x1) ∈ Dec(=(x, x1)) of
      yes(refl(_)) ⇒
        =subst2(
          =cong1(
            appP_T(sb),
            =symm(=trans(=T_disj_vars(sb', disj:(disj∏(dom(sb')), ∉vars_to∉dom(sb', h4))),
                          =symm(=TappP-S(sb', var(x1), h2)))),
          =cong1(appP_T(sb), =varappP(x1, t, sb')),
          h1)
      no(h) ⇒ =cong1(appP_T(sb), ≠varappP(t, sb', h))
    end
  =∧idem_to=T(fun(f, lt), h, h1) ≡ =cong1(fun(f), =∧idem_to=VT(lt, h, h1))
=∧idem_to=VT ∈ (lt ∈ VTerm(n);
                 Idem(:(sb', .(x, t)));
                 =(appP_T(sb, var(x)), appP_T(sb, t))
                ) =(appP_VT(sb, appP_VT(sb', lt)), appP_VT(sb, appP_VT(:(sb', .(x, t)), lt)))
  =∧idem_to=VT([]v, h, h1) ≡ refl(appP_VT(sb, appP_VT(sb', []v)))
  =∧idem_to=VT(:v(lt', t'), h, h1) ≡ =cong2(:v, =∧idem_to=VT(lt', h, h1), =∧idem_to=T(t', h, h1))
idem:= ∈ (Idem(sb); ∉L(x, vars_T(t)); Disjoint(vars_T(t), dom(sb))) Idem(:=S(x, t, sb))
  idem:=([]idem, h1, h2) ≡ []idem
  idem:=(:idem(h3, h4, h5), h1, h2) ≡
    :idem(idem:=(h3, h1, disj_redR(h2)),
          disj:R(:=Tdisj:=S(x, sb1, disj_redR(h2), disj_redR(h4)),
                 ∉:=T≠var(x, t1, disj_to∉(h2), disj_to∉(h4))),
          ∉:=S≠var(x, sb1, disj_to∉(h2), h5))
idem: ∈ (∉L(x, vars_T(t)); ∉L(x, vars_S(sb)); Idem(sb)) Idem(:(sb, .(x, appP_T(sb, t))))
  idem:(h, h1, h2) ≡
    :idem(h2,
          disj:R(=subst1(=symm(=TappP-S(sb, t, h2)), disj_vars_appS(t, h2)),
                 =subst1(=symm(=TappP-S(sb, t, h2)), ∉vars_appS(sb, h, h1))),
          h1)
=var_termappP_T ∈ (∉L(x, vars_T(t));
                   ∉L(x, vars_S(sb));
                   Idem(sb)
                  ) =(appP_T(:(sb, .(x, appP_T(sb, t))), var(x)),
                      appP_T(:(sb, .(x, appP_T(sb, t))), t))
  =var_termappP_T(h, h1, h2) ≡
    =subst2(=trans(=symm(=varappP(x, appP_T(sb, t), sb)), =TappP-S(sb, t, h2)),
            =symm(=TappP-S(:(sb, .(x, appP_T(sb, t))), t, idem:(h, h1, h2))),
            =cong1(appS_T(sb), =:=T(appP_T(sb, t), t, h)))
```

```
∈∧≠∧unify_to⊥ ∈ (t ∈ Term;
                 ∈_L(x, vars_T(t));
                 ¬(=(var(x), t));
                 =(appP_T(sb, var(x)), appP_T(sb, t))
                ) ⊥
  ∈∧≠∧unify_to⊥(var(x1), ∈_hd(_, _), ⇒_I(f1), h2) ≡ f1(refl(var(x1)))
  ∈∧≠∧unify_to⊥(var(x1), ∈_tl(_, _, h3), h1, h2) ≡ case h3 ∈ ∈_L(x, []) of
                                                   end
  ∈∧≠∧unify_to⊥(fun(f, lt), h, h1, h2) ≡
    <∧=_to⊥(≤_to<_sL(∈_to≤_#funsVT(sb, lt, h)), =_cong1(#funs_T, h2))
unifies_S_:=_L ∈ (sb' ∈ Subst;
                  =(appP_T(sb, var(x)), appP_T(sb, t));
                  unifies_S(sb, :=_S(x, t, sb'))
                 ) unifies_S(sb, sb')
  unifies_S_:=_L([], h, h1) ≡ unifies_S_[](sb)
  unifies_S_:=_L(:(sb1, .(y, t1)), h, unifies_S_:(h2, h3)) ≡
    unifies_S_:(unifies_S_:=_L(sb1, h, h2), =_trans(h3, =_symm(=:=_appP_T(t1, h))))
unifies_S_:=_R ∈ (sb' ∈ Subst;
                  =(appP_T(sb, var(x)), appP_T(sb, t));
                  unifies_S(sb, sb')
                 ) unifies_S(sb, :=_S(x, t, sb'))
  unifies_S_:=_R([], h, h1) ≡ unifies_S_[](sb)
  unifies_S_:=_R(:(sb1, .(y, t1)), h, unifies_S_:(h2, h3)) ≡
    unifies_S_:(unifies_S_:=_R(sb1, h, h2), =_trans(h3, =:=_appP_T(t1, h)))
unifies_S_:_:=_R ∈ (=(appP_T(sb, var(x)), appP_T(sb, t));
                    unifies_S(sb, sb')
                   ) unifies_S(sb, :(:=_S(x, t, sb'), .(x, t)))
  unifies_S_:_:=_R(h, h1) ≡ unifies_S_:(unifies_S_:=_R(sb', h, h1), h)
unifies_to unifies ∈ (unifies_S(sb, sb');
                      Idem(:(sb, .(x, t)));
                      ∉_L(x, vars_S(sb'))
                     ) unifies_S(:(sb, .(x, t)), sb')
  unifies_to unifies(unifies_S_[](_), h1, h2) ≡ unifies_S_[](:(sb, .(x, t)))
  unifies_to unifies(unifies_S_:(h3, h4), :_idem(h, h5, h6), ∉:(h1, h7)) ≡
    unifies_S_:(unifies_to unifies(h3, :_idem(h, h5, h6), ∉++_redR(vars_T(t1), h1)),
      =_subst2(=_symm(≠_var appP(t, sb, h7)),
        =_subst2(=_T appP-S(sb, t1, h),
          =_symm(=_T appP-S(:(sb, .(x, t)), t1, :_idem(h, h5, h6))),
          =_cong1(appS_T(sb), =_:=T(t, t1, ∉++_redL(vars_T(t1), h1)))),
        h4))
idem_to unifies ∈ (Idem(sb)) unifies_S(sb, sb)
  idem_to unifies([]_idem) ≡ unifies_S_[]([])
  idem_to unifies(:_idem(h1, h2, h3)) ≡
    unifies_S_:(unifies_to unifies(idem_to unifies(h1), :_idem(h1, h2, h3), h3),
      =_subst2(=_symm(=_var appP(x, t, sb1)),
        =_symm(=_T appP-S(:(sb1, .(x, t)), t, :_idem(h1, h2, h3))),
        =_T disj_vars(:(sb1, .(x, t)), h2)))
```

$\text{unifies}_{\wedge\text{idem}_{to}\text{mgu}_{aux}} \in (\text{Idem}(sb'); \text{unifies}_S(sb, sb')) \leq_{Sb}(sb', sb)$

$\text{unifies}_{\wedge\text{idem}_{to}\text{mgu}_{aux}}([]_{idem}, h_1) \equiv$

$\exists_I(sb, \forall_I([t]=_{cong1}(\text{appP}_T(sb), =_{symm}(=_T\text{appP-S}([], t, []_{idem})))))$

$\text{unifies}_{\wedge\text{idem}_{to}\text{mgu}_{aux}}(:_{idem}(h_2, h_3, h_4), \text{unifies}_{S\_:}(h, h_5)) \equiv$

case $\text{unifies}_{\wedge\text{idem}_{to}\text{mgu}_{aux}}(h_2, h) \in \exists(\text{Subst}, [sb_{1'}]$ **of**

$\forall(\text{Term}, [t']=(\text{appP}_T(sb, t'), \text{appP}_T(sb_{1'}, \text{appP}_T(sb_1, t')))))$

$\exists_I(sb_2, \forall_I(h_1)) \Rightarrow$

$\exists_I(sb_2,$

$\forall_I([t']=_{trans}($

$h_1(t'),$

$=_{\wedge\text{idem}_{to}=_T}($

$t',$

$:_{idem}(h_2, h_3, h_4),$

$=_{subst2}($

$=_{trans}($

$\text{idem}_{\wedge\text{disj}_{to}=(sb_2, h_2, \notin_{to}\text{disj}(\notin_{vars_{to}}\notin_{dom}(sb_1, h_4)))},$

$=_{symm}(h_1(\text{var}(x)))),$

$=_{trans}(h_1(t), =_{symm}(\text{idem}_{\wedge\text{disj}_{to}=(sb_2, h_2, \text{disj}_{redR}(h_3))}),$

$h_5)))))$

end

C.5 Unification Algorithm

C.5.1 Unification Algorithm using the Accessibility Predicate

$<_{LPT} \in (lp_1, lp_2 \in \text{ListPT})$ **Set**

$<_{LPT} \in (<_{N3}(\text{LPT}_{to}\text{N3}(lp_1), \text{LPT}_{to}\text{N3}(lp_2))) <_{LPT}(lp_1, lp_2)$

$<_{\text{LPT}_{to}\text{N3}} \in (<_{LPT}(lp_1, lp_2)) <_{N3}(\text{LPT}_{to}\text{N3}(lp_1), \text{LPT}_{to}\text{N3}(lp_2))$

$<_{\text{LPT}_{to}\text{N3}}(<_{LPT}(h_1)) \equiv h_1$

$\text{acc}_{LPT\_aux} \in (\text{Acc}(\text{N3}, <_{N3}, \text{LPT}_{to}\text{N3}(lp))) \text{ Acc}(\text{ListPT}, <_{LPT}, lp)$

$\text{acc}_{LPT\_aux}(\text{acc}(\_, h_1)) \equiv \text{acc}(lp, [lp', h]\text{acc}_{LPT\_aux}(h_1(\text{LPT}_{to}\text{N3}(lp'), <_{\text{LPT}_{to}\text{N3}}(h))))$

$\text{allacc}_{LPT} \in (lp \in \text{ListPT}) \text{ Acc}(\text{ListPT}, <_{LPT}, lp)$

$\text{allacc}_{LPT}(lp) \equiv \text{acc}_{LPT\_aux}(\text{allacc}_{N3}(\text{LPT}_{to}\text{N3}(lp)))$

$\text{WF}_{LPT} \in \text{WF}(\text{ListPT}, <_{LPT})$

$\text{WF}_{LPT} \equiv \forall_I(\text{allacc}_{LPT})$

```
unify_acc ∈ (lp ∈ ListPT; sb ∈ Subst; Acc(ListPT, <_LPT, lp)) ∨(Subst, Error)
    unify_acc([], sb, acc(_, h_1)) ≡ ∨_L(sb)
    unify_acc(:(lp_1, .(var(x), var(x_1))), sb, acc(_, h_1)) ≡
        case Var_dec(x, x_1) ∈ Dec(=(x, x_1)) of
            yes(refl(_)) ⇒ unify_acc(lp_1, sb, h_1(lp_1, <_LPT(<_LPTvar_var(x_1, lp_1))))
            no(h) ⇒
                unify_acc(:=_LPT(x, var(x_1), lp_1),
                          :(:=_S(x, var(x_1), sb), .(x, var(x_1))),
                          h_1(:=_LPT(x, var(x_1), lp_1), <_LPT(<_LPT:=var_term(lp_1, ∉:(∉[](x), h)))))
        end
    unify_acc(:(lp_1, .(var(x), fun(f, lt))), sb, acc(_, h_1)) ≡
        case ∈_dec(x, vars_T(fun(f, lt))) ∈ Dec(∈_L(x, vars_T(fun(f, lt)))) of
            yes(h) ⇒ ∨_R(error)
            no(h) ⇒
                unify_acc(
                    :=_LPT(x, fun(f, lt), lp_1),
                    :(:=_S(x, fun(f, lt), sb), .(x, fun(f, lt))),
                    h_1(:=_LPT(x, fun(f, lt), lp_1), <_LPT(<_LPT:=var_term(lp_1, ¬∈_to∉(vars_T(fun(f, lt)), h)))))
        end
    unify_acc(:(lp_1, .(fun(f, lt), var(x))), sb, acc(_, h_1)) ≡
        unify_acc(:(lp_1, .(var(x), fun(f, lt))),
                  sb,
                  h_1(:(lp_1, .(var(x), fun(f, lt))), <_LPT(<_LPTvar_fun(f, x, lt, lp_1))))
    unify_acc(:(lp_1, .(fun(f_1, lt_1), fun(f_2, lt_2))), sb, acc(_, h_1)) ≡
        case Fun_dec(f_1, f_2) ∈ Dec(=(f_1, f_2)) of
            yes(refl(_)) ⇒
                case N_dec(n_1, n_2) ∈ Dec(=(n_1, n_2)) of
                    yes(refl(_)) ⇒
                        unify_acc(++(zip(lt_1, lt_2), lp_1),
                                  sb,
                                  h_1(++(zip(lt_1, lt_2), lp_1), <_LPT(<_LPTzip_fun_fun(f_2, f_2, lt_1, lt_2, lp_1))))
                    no(h_2) ⇒ ∨_R(error)
                end
            no(h) ⇒ ∨_R(error)
        end
Unify_acc ∈ (lp ∈ ListPT) ∨(Subst, Error)
    Unify_acc(lp) ≡ unify_acc(lp, [], allacc_LPT(lp))
```

C.5.2 The UniAcc Predicate

UniAcc $\in$ ($lp \in$ ListPT) **Set**

$\text{uniacc}_{[]} \in$ UniAcc([])

$\text{uniacc}_{\text{var\_var}} \in$ ($x \in$ Var; UniAcc(lp)) UniAcc(:(lp, .(var(x), var(x))))

$\text{uniacc}_{\text{var\_term}} \in$ ($lp \in$ ListPT; $\in_{\text{L}}(x, \text{vars}_{\text{T}}(t))$; $\neg$(=(var(x), t))) UniAcc(:(lp, .(var(x), t)))

$\text{uniacc}_{:=\text{var\_term}} \in$ ($\notin_{\text{L}}(x, \text{vars}_{\text{T}}(t))$; UniAcc($:=_{\text{LPT}}(x, t, lp)$)) UniAcc(:($lp$, .(var($x$), t)))

$\text{uniacc}_{\text{var\_fun}} \in$ (UniAcc(:(lp, .(var(x), fun(f, lt))))) UniAcc(:(lp, .(fun(f, lt), var(x))))

$\text{uniacc}_{\text{fun\_fun}} \in$ ($lt_1 \in$ VTerm(n_1);
$lt_2 \in$ VTerm(n_2);
$lp \in$ ListPT;
$\vee(\neg(=(f, g)), \neg(=(n_1, n_2)))$
) UniAcc(:(lp, .(fun(f, lt_1), fun(g, lt_2))))

$\text{uniacc}_{\text{zip\_fun\_fun}} \in$ ($f \in$ Fun;
UniAcc(++(zip(lt_1, lt_2), lp))
) UniAcc(:(lp, .(fun(f, lt_1), fun(f, lt_2))))

$\text{uniacc}_{\text{aux}} \in$ (Acc(N3, $<_{\text{N3}}$, n_3);
$f \in$ ($m_3 \in$ N3; $<_{\text{N3}}(m_3, n_3)$) $\forall$(ListPT, [lp]$\Rightarrow$(=($\text{LPT}_{\text{to}}\text{N3}(lp)$, m_3), UniAcc(lp)))
) $\forall$(ListPT, [lp]$\Rightarrow$(=($\text{LPT}_{\text{to}}\text{N3}(lp)$, n_3), UniAcc(lp)))

$\text{uniacc}_{\text{aux}}(h, f) \equiv \forall_{\text{I}}([lp]\Rightarrow_{\text{I}}([h']\text{uniacc}_{\text{aux2}}(h, f, lp, h')))$

$\text{P}_{\text{n}} \in$ ($n_3 \in$ N3) $\forall$(ListPT, [lp]$\Rightarrow$(=($\text{LPT}_{\text{to}}\text{N3}(lp)$, n_3), UniAcc(lp)))

$\text{P}_{\text{n}}(n_3) \equiv$ wfrec(n_3, $\text{allacc}_{\text{N3}}(n_3)$, $\text{uniacc}_{\text{aux}}$)

$\text{allUniAcc}_{\text{LPT}} \in$ ($lp \in$ ListPT) UniAcc(lp)

$\text{allUniAcc}_{\text{LPT}}(lp) \equiv \Rightarrow_{\text{E}}(\forall_{\text{E}}(\text{P}_{\text{n}}(\text{LPT}_{\text{to}}\text{N3}(lp)), lp), \text{refl}(\text{LPT}_{\text{to}}\text{N3}(lp)))$

$\mathrm{uniacc}_{\mathrm{aux2}} \in (\mathrm{Acc}(\mathrm{N3}, <_{\mathrm{N3}}, n_3);$
$f \in (m_3 \in \mathrm{N3}; <_{\mathrm{N3}}(m_3, n_3)) \forall(\mathrm{ListPT}, [lp'] \Rightarrow (=(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(lp'), m_3), \mathrm{UniAcc}(lp')));$
$lp \in \mathrm{ListPT};$
$=(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(lp), n_3)$
$)\,\mathrm{UniAcc}(lp)$

$\mathrm{uniacc}_{\mathrm{aux2}}(p, f, [], h) \equiv \mathrm{uniacc}_{[]}$

$\mathrm{uniacc}_{\mathrm{aux2}}(p, f, :(lp_1, .(\mathrm{var}(x), \mathrm{var}(x_1))), \mathrm{refl}(\_)) \equiv$
case $\mathrm{Var}_{\mathrm{dec}}(x, x_1) \in \mathrm{Dec}(=(x, x_1))$ **of**
yes(refl(_)) $\Rightarrow$
$\mathrm{uniacc}_{\mathrm{var\_var}}(x_1, \Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(f(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(lp_1), <_{\mathrm{LPTvar\_var}}(x_1, lp_1)), lp_1), \mathrm{refl}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(lp_1))))$
no(h) $\Rightarrow$
$\mathrm{uniacc}_{:=\mathrm{var\_term}}(\notin_{:(\notin_{[]}(x)}, h),$
$\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(f(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:=_{\mathrm{LPT}}(x, \mathrm{var}(x_1), lp_1)), <_{\mathrm{LPT}:=\mathrm{var\_term}}(lp_1, \notin_{:(\notin_{[]}(x)}, h))),$
$:=_{\mathrm{LPT}}(x, \mathrm{var}(x_1), lp_1)),$
$\mathrm{refl}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:=_{\mathrm{LPT}}(x, \mathrm{var}(x_1), lp_1)))))$
end

$\mathrm{uniacc}_{\mathrm{aux2}}(p, f, :(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f_1, lt_1))), \mathrm{refl}(\_)) \equiv$
case $\in_{\mathrm{dec}}(x, \mathrm{vars}_{\mathrm{VT}}(lt_1)) \in \mathrm{Dec}(\in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{VT}}(lt_1)))$ **of**
yes(h) $\Rightarrow$ $\mathrm{uniacc}_{\mathrm{var\_term}}(lp_1, h, \neq_{\mathrm{T}}(f_1, x, lt_1))$
no(h) $\Rightarrow$
$\mathrm{uniacc}_{:=\mathrm{var\_term}}(\neg\in_{\mathrm{to}}\notin(\mathrm{vars}_{\mathrm{T}}(\mathrm{fun}(f_1, lt_1)), h),$
$\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(f(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:=_{\mathrm{LPT}}(x, \mathrm{fun}(f_1, lt_1), lp_1)),$
$<_{\mathrm{LPT}:=\mathrm{var\_term}}(lp_1, \neg\in_{\mathrm{to}}\notin(\mathrm{vars}_{\mathrm{T}}(\mathrm{fun}(f_1, lt_1)), h))),$
$:=_{\mathrm{LPT}}(x, \mathrm{fun}(f_1, lt_1), lp_1)),$
$\mathrm{refl}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:=_{\mathrm{LPT}}(x, \mathrm{fun}(f_1, lt_1), lp_1)))))$
end

$\mathrm{uniacc}_{\mathrm{aux2}}(p, f, :(lp_1, .(\mathrm{fun}(f_1, lt_1), \mathrm{var}(x))), \mathrm{refl}(\_)) \equiv$
$\mathrm{uniacc}_{\mathrm{var\_fun}}(\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(f(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f_1, lt_1)))), <_{\mathrm{LPTvar\_fun}}(f_1, x, lt_1, lp_1)),$
$:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f_1, lt_1)))),$
$\mathrm{refl}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(:(lp_1, .(\mathrm{var}(x), \mathrm{fun}(f_1, lt_1)))))))$

$\mathrm{uniacc}_{\mathrm{aux2}}(p, f, :(lp_1, .(\mathrm{fun}(f_1, lt_1), \mathrm{fun}(f_2, lt_2))), \mathrm{refl}(\_)) \equiv$
case $\mathrm{Fun}_{\mathrm{dec}}(f_1, f_2) \in \mathrm{Dec}(=(f_1, f_2))$ **of**
yes(refl(_)) $\Rightarrow$
case $\mathrm{N}_{\mathrm{dec}}(n_1, n_2) \in \mathrm{Dec}(=(n_1, n_2))$ **of**
yes(refl(_)) $\Rightarrow$
$\mathrm{uniacc}_{\mathrm{zip\_fun\_fun}}($
$f_2,$
$\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(f(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(++(\mathrm{zip}(lt_1, lt_2), lp_1)), <_{\mathrm{LPTzip\_fun\_fun}}(f_2, f_2, lt_1, lt_2, lp_1)),$
$++(\mathrm{zip}(lt_1, lt_2), lp_1)),$
$\mathrm{refl}(\mathrm{LPT}_{\mathrm{to}}\mathrm{N3}(++(\mathrm{zip}(lt_1, lt_2), lp_1)))))$
no(h_1) $\Rightarrow$ $\mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\mathrm{R}}(h_1))$
end
no(h) $\Rightarrow$ $\mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\mathrm{L}}(h))$
end

C.5.3 Unification Algorithm using the UniAcc Predicate

unify $\in$ ($sb \in$ Subst; UniAcc(lp)) $\vee$(Subst, Error)
  unify(sb, uniacc$_{[]}$) $\equiv$ $\vee_L$(sb)
  unify(sb, uniacc$_{var_var}$(x, h_1)) $\equiv$ unify(sb, h_1)
  unify(sb, uniacc$_{var_term}$(lp_1, h_1, h_2)) $\equiv$ $\vee_R$(error)
  unify(sb, uniacc$_{:=var_term}$(h_1, h_2)) $\equiv$ unify(:(:=$_S$(x, t, sb), .(x, t)), h_2)
  unify(sb, uniacc$_{var_fun}$(h_1)) $\equiv$ unify(sb, h_1)
  unify(sb, uniacc$_{fun_fun}$(lt_1, lt_2, lp_1, h_1)) $\equiv$ $\vee_R$(error)
  unify(sb, uniacc$_{zip_fun_fun}$(f, h_1)) $\equiv$ unify(sb, h_1)
Unify $\in$ ($lp \in$ ListPT) $\vee$(Subst, Error)
  Unify(lp) $\equiv$ unify([], allUniAcc$_{LPT}$(lp))

C.5.4 Properties of the Unification Algorithm

unifies$\wedge$error$_{to}\bot$ $\in$ ($p \in$ UniAcc(lp);
  $\exists$(Subst, [sb']unifies$_{LPT}$(sb', lp));
  =(unify(sb, p), $\vee_R$(error))
)$\bot$
  unifies$\wedge$error$_{to}\bot$(uniacc$_{[]}$, h, h_1) $\equiv$ **case** $h_1 \in$ =(unify(sb, uniacc$_{[]}$), $\vee_R$(error)) **of**
    end
  unifies$\wedge$error$_{to}\bot$(uniacc$_{var_var}$(x, h_2), $\exists_I$(sb_1, h), h_1) $\equiv$
    unifies$\wedge$error$_{to}\bot$(h_2, $\exists_I$(sb_1, unifies$_{LPT_red}$(h)), h_1)
  unifies$\wedge$error$_{to}\bot$(uniacc$_{var_term}$(lp_1, h_2, h_3), $\exists_I$(sb_1, h), h_1) $\equiv$ unifies$_{LPT\wedge\in to}\bot$(h, h_2, h_3)
  unifies$\wedge$error$_{to}\bot$(uniacc$_{:=var_term}$(h_2, h_3), $\exists_I$(sb_1, h), h_1) $\equiv$
    unifies$\wedge$error$_{to}\bot$(h_3, $\exists_I$(sb_1, unifies$_{LPT_:=R}$(h)), h_1)
  unifies$\wedge$error$_{to}\bot$(uniacc$_{var_fun}$(h_2), $\exists_I$(sb_1, h), h_1) $\equiv$
    unifies$\wedge$error$_{to}\bot$(h_2, $\exists_I$(sb_1, unifies$_{LPT_fv_{to}vf}$(h)), h_1)
  unifies$\wedge$error$_{to}\bot$(uniacc$_{fun_fun}$(lt_1, lt_2, lp_1, $\vee_L$($\Rightarrow_I$(f_1))), $\exists_I$(sb_1, h), h_1) $\equiv$ f_1(unifies$_{LPTto=f}$(h))
  unifies$\wedge$error$_{to}\bot$(uniacc$_{fun_fun}$(lt_1, lt_2, lp_1, $\vee_R$($\Rightarrow_I$(f_1))), $\exists_I$(sb_1, h), h_1) $\equiv$ f_1(unifies$_{LPTto=arity}$(h))
  unifies$\wedge$error$_{to}\bot$(uniacc$_{zip_fun_fun}$(f, h_2), $\exists_I$(sb_1, h), h_1) $\equiv$
    unifies$\wedge$error$_{to}\bot$(h_2, $\exists_I$(sb_1, unifies$_{LPT_fun_{to}zip}$(h)), h_1)

```
vars_lemma ∈ (p ∈ UniAcc(lp);
               =(unify(sb, p), ∨_L(sb'));
               ⊆(vars_LPT(lp), l);
               ⊆(vars_S(sb), l)
              ) ⊆(vars_S(sb'), l)
    vars_lemma(uniacc_[], refl(_), h1, h2) ≡ h2
    vars_lemma(uniacc_var_var(x, h3), h, h1, h2) ≡ vars_lemma(h3, h, ⊆_trans(⊆vars_var_var(x, lp1), h1), h2)
    vars_lemma(uniacc_var_term(lp1, h3, h4), h, h1, h2) ≡
        case h ∈ =(unify(sb, uniacc_var_term(lp1, h3, h4)), ∨_L(sb')) of
        end
    vars_lemma(uniacc_:=_var_term(h3, h4), h, h1, h2) ≡
        vars_lemma(h4,
                   h,
                   ⊆vars_:=_LPT(x, lp1, ⊆++_redL(⊆++_redL(h1)), ⊆++_redR(++(vars_T(var(x)), vars_T(t)), h1)),
                   ⊆_:(⊆++_L(⊆vars_:=_S(x, sb, ⊆++_redL(⊆++_redL(h1)), h2), ⊆++_redL(⊆++_redL(h1))),
                       ⊆∈_trans(⊆++_redR(vars_T(t), ⊆++_redL(h1)), ∈_hd(x, []))))
    vars_lemma(uniacc_var_fun(h3), h, h1, h2) ≡
        vars_lemma(h3, h, ⊆_trans(fst(⊆vars_var_fun(f, x, lt, lp1)), h1), h2)
    vars_lemma(uniacc_fun_fun(lt1, lt2, lp1, h3), h, h1, h2) ≡
        case h ∈ =(unify(sb, uniacc_fun_fun(lt1, lt2, lp1, h3)), ∨_L(sb')) of
        end
    vars_lemma(uniacc_zip_fun_fun(f, h3), h, h1, h2) ≡
        vars_lemma(h3, h, ⊆_trans(fst(⊆vars_fun_fun(f, f, lt1, lt2, lp1)), h1), h2)
```

$$
\begin{aligned}
&\mathrm{idem}_{\mathrm{lemma}} \in (p \in \mathrm{UniAcc}(lp);\\
&\quad =(\mathrm{unify}(sb, p), \vee_{\mathrm{L}}(sb'));\\
&\quad \mathrm{Disjoint}(\mathrm{vars}_{\mathrm{LPT}}(lp), \mathrm{dom}(sb));\\
&\quad \mathrm{Idem}(sb)\\
&\quad) \mathrm{Idem}(sb')\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{[]}, \mathrm{refl}(\_), h_1, h_2) \equiv h_2\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_var}}(x, h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(h_3, h, \subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq \mathrm{vars}_{\mathrm{var\_var}}(x, lp_1), h_1), h_2)\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_term}}(lp_1, h_3, h_4), h, h_1, h_2) \equiv\\
&\qquad \textbf{case } h \in =(\mathrm{unify}(sb, \mathrm{uniacc}_{\mathrm{var\_term}}(lp_1, h_3, h_4)), \vee_{\mathrm{L}}(sb')) \textbf{ of}\\
&\qquad \textbf{end}\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{:=\mathrm{var\_term}}(h_3, h_4), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(\\
&\qquad\quad h_4,\\
&\qquad\quad h,\\
&\qquad\quad \mathrm{disj}_{:\mathrm{R}}(=_{\mathrm{subst1}}(=_{\mathrm{dom}}(x, t, sb), \subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq \mathrm{vars}_{\mathrm{var\_term}}(x, t, lp_1), h_1)), \notin :=_{\mathrm{LPT}}=_{\mathrm{var}}(lp_1, h_3)),\\
&\qquad\quad :_{\mathrm{idem}}(\mathrm{idem}_{:=}(h_2, h_3, \subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq\!\!+\!\!+_{\mathrm{RR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1)),\\
&\qquad\qquad \mathrm{disj}_{:\mathrm{R}}(=_{\mathrm{subst1}}(=_{\mathrm{dom}}(x, t, sb),\\
&\qquad\qquad\qquad \subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq\!\!+\!\!+_{\mathrm{RR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1)),\\
&\qquad\qquad\quad h_3),\\
&\qquad\qquad \notin :=_{\mathrm{S}}=_{\mathrm{var}}(\\
&\qquad\qquad\quad sb,\\
&\qquad\qquad\quad h_3,\\
&\qquad\qquad\quad \mathrm{disj}_{\mathrm{to}}\notin(\mathrm{disj}_{\mathrm{symm}}(\subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\subseteq\!\!+\!\!+_{\mathrm{LR}}(\mathrm{vars}_{\mathrm{LPT}}(lp_1), \mathrm{vars}_{\mathrm{T}}(\mathrm{var}(x)), \mathrm{vars}_{\mathrm{T}}(t)), h_1))))))\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{var\_fun}}(h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(h_3, h, \subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\mathrm{fst}(\subseteq \mathrm{vars}_{\mathrm{var\_fun}}(f, x, lt, lp_1)), h_1), h_2)\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, h_3), h, h_1, h_2) \equiv\\
&\qquad \textbf{case } h \in =(\mathrm{unify}(sb, \mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, h_3)), \vee_{\mathrm{L}}(sb')) \textbf{ of}\\
&\qquad \textbf{end}\\
&\quad \mathrm{idem}_{\mathrm{lemma}}(\mathrm{uniacc}_{\mathrm{zip\_fun\_fun}}(f, h_3), h, h_1, h_2) \equiv\\
&\qquad \mathrm{idem}_{\mathrm{lemma}}(h_3, h, \subseteq\!\!\wedge \mathrm{disj}_{\mathrm{to}}\mathrm{disj}(\mathrm{fst}(\subseteq \mathrm{vars}_{\mathrm{fun\_fun}}(f, f, lt_1, lt_2, lp_1)), h_1), h_2)
\end{aligned}
$$

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}} \in (p \in \text{UniAcc}(lp);$
$\text{unifies}_{\text{LPT}}(sb', lp);$
$\text{unifies}_{\text{S}}(sb', sb);$
$=(\text{unify}(sb, p), \vee_{\text{L}}(sb_1))$
$)\,\text{unifies}_{\text{S}}(sb', sb_1)$

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{[]}, h, h_1, \text{refl}(\_)) \equiv h_1$

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{\text{var\_var}}(x, h_3), \text{unifies}_{\text{LPT\_:}}(h_4, h_5), h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(h_3, h_4, h_1, h_2)$

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{\text{var\_term}}(lp_1, h_3, h_4), h, h_1, h_2) \equiv$
case $h_2 \in =(\text{unify}(sb, \text{uniacc}_{\text{var\_term}}(lp_1, h_3, h_4)), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{:=\text{var\_term}}(h_3, h_4), \text{unifies}_{\text{LPT\_:}}(h_5, h_6), h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(h_4, \text{unifies}_{\text{LPT\_:=R}}(\text{unifies}_{\text{LPT\_:}}(h_5, h_6)), \text{unifies}_{\text{S\_:\_:=R}}(h_6, h_1), h_2)$

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{\text{var\_fun}}(h_3), h, h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(h_3, \text{unifies}_{\text{LPT\_fvtovf}}(h), h_1, h_2)$

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{L}}(\Rightarrow_{\text{I}}(f_1))), h, h_1, h_2) \equiv$
case $h_2 \in =(\text{unify}(sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{L}}(\Rightarrow_{\text{I}}(f_1)))), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{R}}(\Rightarrow_{\text{I}}(f_1))), h, h_1, h_2) \equiv$
case $h_2 \in =(\text{unify}(sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, \vee_{\text{R}}(\Rightarrow_{\text{I}}(f_1)))), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(\text{uniacc}_{\text{zip\_fun\_fun}}(f, h_3), h, h_1, h_2) \equiv$
$\text{unifies}_{\text{LpSbtounifies}_{\text{Sb}}}(h_3, \text{unifies}_{\text{LPT\_funtozip}}(h), h_1, h_2)$

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}} \in (p \in \text{UniAcc}(lp);$
$\text{unifies}_{\text{S}}(sb', sb_1);$
$=(\text{unify}(sb, p), \vee_{\text{L}}(sb_1))$
$) \wedge(\text{unifies}_{\text{LPT}}(sb', lp), \text{unifies}_{\text{S}}(sb', sb))$

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{[]}, h, \text{refl}(\_)) \equiv \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_[]}}(sb'), h)$

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{\text{var\_var}}(x, h_2), h, h_1) \equiv$
case $\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(h_2, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', lp_1), \text{unifies}_{\text{S}}(sb', sb))$ **of**
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_:}}(h_3, \text{refl}(\text{appP}_{\text{T}}(sb', \text{var}(x)))), h_4)$
end

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{\text{var\_term}}(lp_1, h_2, h_3), h, h_1) \equiv$
case $h_1 \in =(\text{unify}(sb, \text{uniacc}_{\text{var\_term}}(lp_1, h_2, h_3)), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{:=\text{var\_term}}(h_2, h_3), h, h_1) \equiv$
case $\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(h_3, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', :=_{\text{LPT}}(x, t, lp_1)),$ **of**
$\text{unifies}_{\text{S}}(sb', :(:=_{\text{S}}(x, t, sb), .(x, t))))$
$\wedge_{\text{I}}(h_4, \text{unifies}_{\text{S\_:}}(h_6, h_7)) \Rightarrow$
$\wedge_{\text{I}}(\text{unifies}_{\text{LPT\_:}}(\text{unifies}_{\text{LPT\_:=L}}(lp_1, h_7, h_4), h_7), \text{unifies}_{\text{S\_:=L}}(sb, h_7, h_6))$
end

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{\text{var\_fun}}(h_2), h, h_1) \equiv$
case $\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(h_2, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', :(lp_1, .(\text{var}(x), \text{fun}(f, lt)))),$ **of**
$\text{unifies}_{\text{S}}(sb', sb))$
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_vf}_{\text{to}}\text{fv}}(h_3), h_4)$
end

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, h_2), h, h_1) \equiv$
case $h_1 \in =(\text{unify}(sb, \text{uniacc}_{\text{fun\_fun}}(lt_1, lt_2, lp_1, h_2)), \vee_{\text{L}}(sb_1))$ **of**
end

$\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{uniacc}_{\text{zip\_fun\_fun}}(f, h_2), h, h_1) \equiv$
case $\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(h_2, h, h_1) \in \wedge(\text{unifies}_{\text{LPT}}(sb', ++(\text{zip}(lt_1, lt_2), lp_1)),$ **of**
$\text{unifies}_{\text{S}}(sb', sb))$
$\wedge_{\text{I}}(h_3, h_4) \Rightarrow \wedge_{\text{I}}(\text{unifies}_{\text{LPT\_zip}_{\text{to}}\text{fun}}(f, h_3), h_4)$
end

C.6 External Approach

$\text{error}_{\wedge}\text{unifies}_{\text{to}\perp} \in (=(\text{Unify}(lp), \vee_{\text{R}}(\text{error})); \exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp))) \perp$
$\text{error}_{\wedge}\text{unifies}_{\text{to}\perp}(h, h_1) \equiv \text{unifies}_{\wedge}\text{error}_{\text{to}\perp}(\text{allUniAcc}_{\text{LPT}}(lp), h_1, h)$

$\text{unifies}_{\text{to}\neg}\text{error} \in (\exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp))) \neg(=(\text{Unify}(lp), \vee_{\text{R}}(\text{error})))$
$\text{unifies}_{\text{to}\neg}\text{error}(h) \equiv \Rightarrow_{\text{I}}([h']\text{error}_{\wedge}\text{unifies}_{\text{to}\perp}(h', h))$

$\text{error}_{\text{to}\neg}\text{unifies} \in (=(\text{Unify}(lp), \vee_{\text{R}}(\text{error}))) \neg(\exists(\text{Subst}, [sb]\text{unifies}_{\text{LPT}}(sb, lp)))$
$\text{error}_{\text{to}\neg}\text{unifies}(h) \equiv \Rightarrow_{\text{I}}(\text{error}_{\wedge}\text{unifies}_{\text{to}\perp}(h))$

$\text{idem}_{\text{prop}} \in (=(\text{Unify}(lp), \vee_{\text{L}}(sb))) \text{Idem}(sb)$
$\text{idem}_{\text{prop}}(h) \equiv \text{idem}_{\text{lemma}}(\text{allUniAcc}_{\text{LPT}}(lp), h, \text{disj}_{[]\text{R}}(\text{vars}_{\text{LPT}}(lp)), []_{\text{idem}})$

$\equiv_{\text{Lp\_Unify}} \in (=(\text{Unify}(lp), \vee_{\text{L}}(sb))) \equiv_{\text{LpSb}}(lp, sb)$
$\equiv_{\text{Lp\_Unify}}(h) \equiv$
$\forall_{\text{I}}([sb_1]$
$\wedge_{\text{I}}(\Rightarrow_{\text{I}}([h_1]\text{unifies}_{\text{LpSb}_{\text{to}}\text{unifies}_{\text{Sb}}}(\text{allUniAcc}_{\text{LPT}}(lp), \text{fst}(h_1), \text{snd}(h_1)), h)),$
$\Rightarrow_{\text{I}}([h_1]\text{unifies}_{\text{Sb}_{\text{to}}\text{unifies}_{\text{LpSb}}}(\text{allUniAcc}_{\text{LPT}}(lp), h_1, h))))$

$\mathrm{mgu}_{\mathrm{prop\_aux}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb)); \mathrm{unifies}_{\mathrm{LPT}}(sb', lp)) \leq_{\mathrm{Sb}} (sb, sb')$
$\mathrm{mgu}_{\mathrm{prop\_aux}}(h, h_1) \equiv$
$\mathrm{unifies}_{\wedge}\mathrm{idem}_{\mathrm{to}}\mathrm{mgu}_{\mathrm{aux}}(\mathrm{idem}_{\mathrm{prop}}(h),$
$\mathrm{unifies}_{\mathrm{LpSbto}}\mathrm{unifies}_{\mathrm{Sb}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), h_1, \mathrm{unifies}_{\mathrm{S\_[]}}(sb'), h))$
$\mathrm{unifies}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb))) \mathrm{unifies}_{\mathrm{LPT}}(sb, lp)$
$\mathrm{unifies}_{\mathrm{prop}}(h) \equiv$
$\mathrm{fst}(\mathrm{unifies}_{\mathrm{Sbto}}\mathrm{unifies}_{\mathrm{LpSb}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), \mathrm{idem}_{\mathrm{to}}\mathrm{unifies}(\mathrm{idem}_{\mathrm{prop}}(h)), h))$
$\mathrm{vars}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb))) \subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}(lp))$
$\mathrm{vars}_{\mathrm{prop}}(h) \equiv \mathrm{vars}_{\mathrm{lemma}}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp), h, \subseteq_{\mathrm{refl}}(\mathrm{vars}_{\mathrm{LPT}}(lp)), \subseteq_{[]}(\mathrm{vars}_{\mathrm{LPT}}(lp)))$
$\mathrm{idempotent}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb))) \mathrm{Idempotent}(sb)$
$\mathrm{idempotent}_{\mathrm{prop}}(h) \equiv \mathrm{idem}_{\mathrm{to}}\mathrm{idempotent}(\mathrm{idem}_{\mathrm{prop}}(h))$
$\mathrm{mgu}_{\mathrm{prop}} \in (=(\mathrm{Unify}(lp), \vee_{\mathrm{L}}(sb))) \mathrm{mgu}(sb, lp)$
$\mathrm{mgu}_{\mathrm{prop}}(h) \equiv \wedge_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{prop}}(h), \forall_{\mathrm{I}}([sb']\Rightarrow_{\mathrm{I}}([h_1]\mathrm{mgu}_{\mathrm{prop\_aux}}(h, h_1))))$

C.7 Integrated Approach

$\mathrm{unifies}_{\mathrm{LPT\_var\_var}^{\mathrm{to}}\bot} \in (\neg_{\mathrm{I}}(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, lp)));$
$\exists(\mathrm{Subst}, [sb']\mathrm{unifies}_{\mathrm{LPT}}(sb', :(lp, .(\mathrm{var}(x), \mathrm{var}(x)))))$
$)\bot$
$\mathrm{unifies}_{\mathrm{LPT\_var\_var}^{\mathrm{to}}\bot}(\Rightarrow_{\mathrm{I}}(f), \exists_{\mathrm{I}}(sb_1, \mathrm{unifies}_{\mathrm{LPT\_:}}(h, h_1))) \equiv f(\exists_{\mathrm{I}}(sb_1, h))$
$\mathrm{unifies}_{\mathrm{LPT\_var\_term}\in^{\mathrm{to}}\bot} \in (\in_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t));$
$\neg_{\mathrm{I}}(=(\mathrm{var}(x), t));$
$\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, :(lp, .(\mathrm{var}(x), t))))$
$)\bot$
$\mathrm{unifies}_{\mathrm{LPT\_var\_term}\in^{\mathrm{to}}\bot}(h, h_1, \exists_{\mathrm{I}}(sb_1, h_2)) \equiv \mathrm{unifies}_{\mathrm{LPT}\wedge\in^{\mathrm{to}}\bot}(h_2, h, h_1)$
$\mathrm{unifies}_{\mathrm{LPT\_var\_term}:=^{\mathrm{to}}\bot} \in (\notin_{\mathrm{L}}(x, \mathrm{vars}_{\mathrm{T}}(t));$
$\neg_{\mathrm{I}}(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, :=_{\mathrm{LPT}}(x, t, lp))));$
$\exists(\mathrm{Subst}, [sb']\mathrm{unifies}_{\mathrm{LPT}}(sb', :(lp, .(\mathrm{var}(x), t))))$
$)\bot$
$\mathrm{unifies}_{\mathrm{LPT\_var\_term}:=^{\mathrm{to}}\bot}(h, \Rightarrow_{\mathrm{I}}(f), \exists_{\mathrm{I}}(sb_1, h_1)) \equiv f(\exists_{\mathrm{I}}(sb_1, \mathrm{unifies}_{\mathrm{LPT\_:=R}}(h_1)))$
$\mathrm{unifies}_{\mathrm{LPT\_var\_fun}^{\mathrm{to}}\bot} \in (\neg_{\mathrm{I}}(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, :(lp, .(\mathrm{var}(x), \mathrm{fun}(f, lt))))));$
$\exists(\mathrm{Subst}, [sb']\mathrm{unifies}_{\mathrm{LPT}}(sb', :(lp, .(\mathrm{fun}(f, lt), \mathrm{var}(x)))))$
$)\bot$
$\mathrm{unifies}_{\mathrm{LPT\_var\_fun}^{\mathrm{to}}\bot}(\Rightarrow_{\mathrm{I}}(f_1), \exists_{\mathrm{I}}(sb_1, h)) \equiv f_1(\exists_{\mathrm{I}}(sb_1, \mathrm{unifies}_{\mathrm{LPT\_fv_{to}vf}}(h)))$
$\mathrm{unifies}_{\mathrm{LPT\_fun\_fun}^{\mathrm{to}}\bot} \in (\vee(\neg(=(f, g)), \neg(=(n_1, n_2)));$
$\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, :(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(g, lt_2)))))$
$)\bot$
$\mathrm{unifies}_{\mathrm{LPT\_fun\_fun}^{\mathrm{to}}\bot}(\vee_{\mathrm{L}}(\Rightarrow_{\mathrm{I}}(f_1)), \exists_{\mathrm{I}}(sb_1, h)) \equiv f_1(\mathrm{unifies}_{\mathrm{LPT}^{\mathrm{to}}=_{\mathrm{f}}}(h))$
$\mathrm{unifies}_{\mathrm{LPT\_fun\_fun}^{\mathrm{to}}\bot}(\vee_{\mathrm{R}}(\Rightarrow_{\mathrm{I}}(f_1)), \exists_{\mathrm{I}}(sb_1, h)) \equiv f_1(\mathrm{unifies}_{\mathrm{LPT}^{\mathrm{to}}=_{\mathrm{arity}}}(h))$
$\mathrm{unifies}_{\mathrm{LPT\_zip\_fun\_fun}^{\mathrm{to}}\bot} \in (\neg_{\mathrm{I}}(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, ++(\mathrm{zip}(lt_1, lt_2), lp))));$
$\exists(\mathrm{Subst}, [sb']\mathrm{unifies}_{\mathrm{LPT}}(sb', :(lp, .(\mathrm{fun}(f, lt_1), \mathrm{fun}(f, lt_2)))))$
$)\bot$
$\mathrm{unifies}_{\mathrm{LPT\_zip\_fun\_fun}^{\mathrm{to}}\bot}(\Rightarrow_{\mathrm{I}}(f_1), \exists_{\mathrm{I}}(sb_1, h)) \equiv f_1(\exists_{\mathrm{I}}(sb_1, \mathrm{unifies}_{\mathrm{LPT\_fun_{to}zip}}(h)))$

$\mathrm{mgu}_{\mathrm{auxT}} \in (t' \in \mathrm{Term};$
$\mathrm{Idem}(:(sb, .(x, \mathrm{appP}_{\mathrm{T}}(sb, t))));$
$\mathrm{unifies}_{\mathrm{LPT}}(sb', :(lp, .(\mathrm{var}(x), t)));$
$\forall(\mathrm{Term}, [t_1]{=}(\mathrm{appP}_{\mathrm{T}}(sb', t_1), \mathrm{appP}_{\mathrm{T}}(sb_1, \mathrm{appP}_{\mathrm{T}}(sb, t_1))))$
$) {=} (\mathrm{appP}_{\mathrm{T}}(sb', t'), \mathrm{appP}_{\mathrm{T}}(sb_1, \mathrm{appP}_{\mathrm{T}}(:(sb, .(x, \mathrm{appP}_{\mathrm{T}}(sb, t))), t')))$

$\mathrm{mgu}_{\mathrm{auxT}}(\mathrm{var}(x_1), h_3, \mathrm{unifies}_{\mathrm{LPT\_:}}(h, h_6), h_2) \equiv$
case $\mathrm{Var}_{\mathrm{dec}}(x, x_1) \in \mathrm{Dec}(=(x, x_1))$ **of**
$\mathrm{yes}(\mathrm{refl}(\_)) \Rightarrow$
$=_{\mathrm{subst2}}(h_6, =_{\mathrm{cong1}}(\mathrm{appP}_{\mathrm{T}}(sb_1), =_{\mathrm{var}}\mathrm{appP}(x_1, \mathrm{appP}_{\mathrm{T}}(sb, t), sb)), \forall_{\mathrm{E}}(h_2, t))$
$\mathrm{no}(h_1) \Rightarrow$
$=_{\mathrm{trans}}(\forall_{\mathrm{E}}(h_2, \mathrm{var}(x_1)), =_{\mathrm{cong1}}(\mathrm{appP}_{\mathrm{T}}(sb_1), =_{\mathrm{symm}}(\not\in_{\mathrm{to}}{=}_{\mathrm{T\_appP}}(\not\in :(\not\in [])(x), h_1), h_3))))$
end

$\mathrm{mgu}_{\mathrm{auxT}}(\mathrm{fun}(f, lt), h, h_1, h_2) \equiv =_{\mathrm{cong1}}(\mathrm{fun}(f), \mathrm{mgu}_{\mathrm{auxVT}}(lt, h, h_1, h_2))$

$\mathrm{mgu}_{\mathrm{auxVT}} \in (lt \in \mathrm{VTerm}(n);$
$\mathrm{Idem}(:(sb, .(x, \mathrm{appP}_{\mathrm{T}}(sb, t))));$
$\mathrm{unifies}_{\mathrm{LPT}}(sb', :(lp, .(\mathrm{var}(x), t)));$
$\forall(\mathrm{Term}, [t_1]{=}(\mathrm{appP}_{\mathrm{T}}(sb', t_1), \mathrm{appP}_{\mathrm{T}}(sb_1, \mathrm{appP}_{\mathrm{T}}(sb, t_1))))$
$) {=} (\mathrm{appP}_{\mathrm{VT}}(sb', lt), \mathrm{appP}_{\mathrm{VT}}(sb_1, \mathrm{appP}_{\mathrm{VT}}(:(sb, .(x, \mathrm{appP}_{\mathrm{T}}(sb, t))), lt)))$

$\mathrm{mgu}_{\mathrm{auxVT}}([]_{\mathrm{v}}, h, h_1, h_2) \equiv \mathrm{refl}(\mathrm{appP}_{\mathrm{VT}}(sb', []_{\mathrm{v}}))$

$\mathrm{mgu}_{\mathrm{auxVT}}(:_{\mathrm{v}}(lt', t'), h, h_1, h_2) \equiv =_{\mathrm{cong2}}(:_{\mathrm{v}}, \mathrm{mgu}_{\mathrm{auxVT}}(lt', h, h_1, h_2), \mathrm{mgu}_{\mathrm{auxT}}(t', h, h_1, h_2))$

$\mathrm{Th} \in (\mathrm{UniAcc}(lp)$
$) \vee(\neg(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, lp))),$
$\exists(\mathrm{Subst}, [sb]\wedge_3(\subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}(lp)), \mathrm{Idem}(sb), \mathrm{mgu}(sb, lp))))$

$\mathrm{Th}(\mathrm{uniacc}_{[]}) \equiv$
$\vee_{\mathrm{R}}(\exists_{\mathrm{I}}([],$
$\wedge_{\mathrm{I3}}(\subseteq_{[]}(\mathrm{vars}_{\mathrm{LPT}}([])),$
$[]_{\mathrm{idem}},$
$\wedge_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_[]}}([]),$
$\forall_{\mathrm{I}}([sb']$
$\Rightarrow_{\mathrm{I}}([h]\exists_{\mathrm{I}}(sb',$
$\forall_{\mathrm{I}}([t]{=}_{\mathrm{cong1}}(\mathrm{appP}_{\mathrm{T}}(sb'), =_{\mathrm{symm}}(=_{\mathrm{T}}\mathrm{appP\text{–}S}([], t, []_{\mathrm{idem}}))))))))))$

$\mathrm{Th}(\mathrm{uniacc}_{\mathrm{var\_var}}(x, h_1)) \equiv$
case $\mathrm{Th}(h_1) \in \vee(\neg(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, lp_1))),$ **of**
$\exists(\mathrm{Subst}, [sb]\wedge_3(\subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}(lp_1)), \mathrm{Idem}(sb), \mathrm{mgu}(sb, lp_1))))$
$\vee_{\mathrm{L}}(h) \Rightarrow \vee_{\mathrm{L}}(\Rightarrow_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_var\_var}}{}_{\mathrm{to}}\bot(h)))$
$\vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb, \wedge_{\mathrm{I3}}(h, h_2, \wedge_{\mathrm{I}}(h_3, h_4)))) \Rightarrow$
$\vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb,$
$\wedge_{\mathrm{I3}}(\subseteq_{\mathrm{trans}}(h, \subseteq\mathrm{vars}_{\mathrm{var\_var}}(x, lp_1)),$
$h_2,$
$\wedge_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_:}}(h_3, \mathrm{refl}(\mathrm{appP}_{\mathrm{T}}(sb, \mathrm{var}(x)))),$
$\forall_{\mathrm{I}}([sb']{\Rightarrow_{\mathrm{I}}}([h_5]{\Rightarrow_{\mathrm{E}}}(\forall_{\mathrm{E}}(h_4, sb'), \mathrm{unifies}_{\mathrm{LPT\_red}}(h_5))))))))$
end

$\mathrm{Th}(\mathrm{uniacc}_{\mathrm{var\_term}}(lp_1, h_1, h_2)) \equiv \vee_{\mathrm{L}}(\Rightarrow_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_var\_term}}{\in}_{\mathrm{to}}\bot(h_1, h_2)))$

```
Th(uniacc:=var_term(h1, h2)) ≡
    case Th(h2) ∈ ∨(¬(∃(Subst, [sb]unifiesLPT(sb, :=LPT(x, t, lp1)))), of
                    ∃(Subst,
                      [sb]∧3(⊆(varsS(sb), varsLPT(:=LPT(x, t, lp1))),
                             Idem(sb),
                             mgu(sb, :=LPT(x, t, lp1)))))
        ∨L(h) ⇒ ∨L(⇒I(unifiesLPT_var_term:=to⊥(h1, h)))
        ∨R(∃I(sb, ∧I3(h, h3, ∧I(h4, h5)))) ⇒
            ∨R(∃I(:(sb, .(x, appPT(sb, t))),
                  ∧I3(⊆++L(⊆trans(h, ⊆vars var_term(x, t, lp1)),
                           ⊆:(⊆trans(⊆vars appP_T(sb, t),
                                     ⊆++L(⊆trans(h, ⊆vars var_term(x, t, lp1)),
                                          ⊆++RR(varsLPT(lp1), varsT(var(x)), varsT(t)))),
                              ∈++monL(varsLPT(lp1), ∈++monR(varsT(t), ∈hd(x, [])))),
                      idem:(h1, ⊆∧∉to∉(h, ∉:=LPT=var(lp1, h1)), h3),
                      ∧I(unifiesLPT_:(
                             unifiesLPT:=tounifiesLPT(h1, ⊆∧∉to∉(h, ∉:=LPT=var(lp1, h1)), h3, h4),
                             =var_termappPT(h1, ⊆∧∉to∉(h, ∉:=LPT=var(lp1, h1)), h3)),
                         ∀I([sb']
                             ⇒I([h6]∃I(witness(⇒E(∀E(h5, sb'), unifiesLPT_:=R(h6))),
                                       ∀I([t']mguauxT(
                                                t',
                                                idem:(h1,
                                                      ⊆∧∉to∉(h, ∉:=LPT=var(lp1, h1)),
                                                      h3),
                                                h6,
                                                proof(⇒E(∀E(h5, sb'), unifiesLPT_:=R(h6))))))))))))
    end
Th(uniaccvar_fun(h1)) ≡
    case Th(h1) ∈ ∨(¬(∃(Subst, [sb]unifiesLPT(sb, :(lp1, .(var(x), fun(f, lt)))))), of
                    ∃(Subst,
                      [sb]∧3(⊆(varsS(sb), varsLPT(:(lp1, .(var(x), fun(f, lt))))),
                             Idem(sb),
                             mgu(sb, :(lp1, .(var(x), fun(f, lt)))))))
        ∨L(h) ⇒ ∨L(⇒I(unifiesLPT_var_funto⊥(h)))
        ∨R(∃I(sb, ∧I3(h, h2, ∧I(h3, h4)))) ⇒
            ∨R(∃I(sb,
                  ∧I3(⊆trans(h, fst(⊆vars var_fun(f, x, lt, lp1))),
                      h2,
                      ∧I(unifiesLPT_vftofv(h3),
                         ∀I([sb']⇒I([h7]⇒E(∀E(h4, sb'), unifiesLPT_fvtovf(h7))))))))
    end
```

$\mathrm{Th}(\mathrm{uniacc}_{\mathrm{fun\_fun}}(lt_1, lt_2, lp_1, h_1)) \equiv \vee_{\mathrm{L}}(\Rightarrow_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_fun\_fun\,to}\bot}(h_1)))$
$\mathrm{Th}(\mathrm{uniacc}_{\mathrm{zip\_fun\_fun}}(f, h_1)) \equiv$
case $\mathrm{Th}(h_1) \in \vee(\neg(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, {+}{+}(\mathrm{zip}(lt_1, lt_2), lp_1)))),$ **of**
$\exists(\mathrm{Subst},$
$[sb]\wedge_3(\subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}({+}{+}(\mathrm{zip}(lt_1, lt_2), lp_1))),$
$\mathrm{Idem}(sb),$
$\mathrm{mgu}(sb, {+}{+}(\mathrm{zip}(lt_1, lt_2), lp_1)))))$
$\vee_{\mathrm{L}}(h) \Rightarrow \vee_{\mathrm{L}}(\Rightarrow_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_zip\_fun\_fun\,to}\bot}(h)))$
$\vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb, \wedge_{\mathrm{I3}}(h, h_2, \wedge_{\mathrm{I}}(h_3, h_4)))) \Rightarrow$
$\vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb,$
$\wedge_{\mathrm{I3}}(\subseteq_{\mathrm{trans}}(h, \mathrm{fst}(\subseteq\mathrm{vars}_{\mathrm{fun\_fun}}(f, f, lt_1, lt_2, lp_1))),$
$h_2,$
$\wedge_{\mathrm{I}}(\mathrm{unifies}_{\mathrm{LPT\_zip\,to\,fun}}(f, h_3),$
$\forall_{\mathrm{I}}([sb']\Rightarrow_{\mathrm{I}}([h_7]\Rightarrow_{\mathrm{E}}(\forall_{\mathrm{E}}(h_4, sb'), \mathrm{unifies}_{\mathrm{LPT\_fun\,to\,zip}}(h_7))))))))$
end
Theorem $\in$ $(lp \in \mathrm{ListPT}$
$) \vee(\neg(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, lp))),$
$\exists(\mathrm{Subst}, [sb]\wedge_3(\subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}(lp)), \mathrm{Idempotent}(sb), \mathrm{mgu}(sb, lp))))$
$\mathrm{Theorem}(lp) \equiv$
case $\mathrm{Th}(\mathrm{allUniAcc}_{\mathrm{LPT}}(lp)) \in \vee(\neg(\exists(\mathrm{Subst}, [sb]\mathrm{unifies}_{\mathrm{LPT}}(sb, lp))),$
$\exists(\mathrm{Subst},$
$[sb]\wedge_3(\subseteq(\mathrm{vars}_{\mathrm{S}}(sb), \mathrm{vars}_{\mathrm{LPT}}(lp)), \mathrm{Idem}(sb), \mathrm{mgu}(sb, lp))))$
$\vee_{\mathrm{L}}(h) \Rightarrow \vee_{\mathrm{L}}(h)$
$\vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb, \wedge_{\mathrm{I3}}(h_1, h_2, h_3))) \Rightarrow \vee_{\mathrm{R}}(\exists_{\mathrm{I}}(sb, \wedge_{\mathrm{I3}}(h_1, \mathrm{idem}_{\mathrm{to}}\mathrm{idempotent}(h_2), h_3)))$
end

Bibliography

[Acz77] P. Aczel. An Introduction to Inductive Definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland Publishing Company, 1977.

[AGNvS94] T. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994. Available on the WWW `ftp://ftp.cs.chalmers.se/pub/users/alti/alf.ps.Z`.

[BM79] R. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[CM81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.

[CNSvS94] T. Coquand, B. Nordström, J. M. Smith, and B. von Sydow. Type theory and programming. *EATCS*, 52, February 1994.

[Coq92] T. Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Coq98] C. Coquand. The homepage of the Agda type checker. Homepage: `http://www.cs.chalmers.se/~catarina/Agda/`, 1998.

[CP90] T. Coquand and C. Paulin. Inductively defined types. In *Proceedings of COLOG-88*, number 417 in Lecture Notes in Computer Science, pages 50–66. Springer-Verlag, 1990.

[Dam85] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, Department of Cimputer Science, University of Edinburgh, April 1985.

[DFH+91] G. Dowek, A. Felty, H. Herbelin, H. Huet, G. P. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The coq proof assistant user's guide version 5.6. Technical report, Rapport Technique 134, INRIA, December 1991.

[Eri84] L-H. Eriksson. Synthesis of a unification algorithm in a logic programming calculus. *Journal of Logic Programming*, 1(1):3–18, 1984.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[Jau97] M. Jaume. Unification : a Case Study in Transposition of Formal Properties. In E.L. Gunter and A. Felty, editors, *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: Poster session TPHOLs'97*, pages 79–93, Murray Hill, N.J., 1997.

[JHe+99] Simon Peyton Jones, John Hughes, (editors), Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS Technical Report ECS-LFCS-92-211, 1992.

[Mag92] L. Magnusson. The new Implementation of ALF. In *The informal proceeding from the logical framework workshop at Båstad, June 1992*, 1992.

[McB99] C. McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, Department of Computer Science, University of Edinburgh, October 1999.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[ML82] P. Martin-Löf. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[MM82] A. Martelli and U. Montanari. An efficient unificaction algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MN94] L. Magnusson and B. Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[MW81] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981. North-Holland Publishing Company.

[Nor88] B. Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pau85] L. Paulson. Verifying the unification algorithmn in LCF. *Science of Computer Programming*, 5:143–169, 1985. North-Holland.

[PJ87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[PM93] C. Paulin-Mohring. Inductive Definitions in the system Coq; rules and properties. In M. Bezem and J. F. Groote, editors, *Proceeding of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 328–345. Springer-Verlag, LNCS 664, March 1993.

[RL] J. Rouyer and P. Lescanne. Verification and programming of first-order unification in the calculus of constructions with inductive types. English summary of [Rou92].

[Rob65] J. A. Robinson. A Machine-oriented Logic Based on the Resolution Principle. *ACM*, 12:23–41, 1965.

[Rou92] J. Rouyer. Développement de l'algorithme d'unification dans le calcul des constructions avec types inductifs. Technical Report 1795, INRIA-Lorraine, November 1992. See also [RL].

[RRAHM99] J. L. Ruiz-Reina, J. A. Alonso, M. J. Hidalgo, and F. J. Martín. Mechanical verification of a rule-based unification algorithm in the Boyer-Moore theorem prover. In *Proceeding of the Conference on Declarative Programming - AGP99, to appear*, September 1999.

[Sza97] N. Szasz. *A Theory of Specifications, Programs and Proofs*. PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Göteborg, Sweden, June 1997.

[wor99] Workshop on dependent types in programming. Available on the WWW `http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html`, 27 - 28 March 1999. Göteborg, Sweden.